
RedisVL

Release 0.11.0

Redis Applied AI

Nov 07, 2025

CONTENTS

1	Installation	3
2	Table of Contents	5
2.1	Overview	5
2.2	RedisVL API	11
2.3	User Guides	141
2.4	Example Gallery	240
	Index	241

A powerful, AI-native Python client library for [Redis](#). Leverage the speed, flexibility, and reliability of Redis for real-time data to supercharge your AI application.

Index Management

Design search schema and indices with ease from YAML, with Python, or from the CLI.

Advanced Vector Search

Perform powerful vector search queries with complex filtering support.

Embedding Creation

Use OpenAI or any of the other supported vectorizers to create embeddings.

CLI

Interact with RedisVL using a Command Line Interface (CLI) for ease of use.

Semantic Caching

Extend RedisVL to cache LLM results, increasing QPS and decreasing system cost.

Example Gallery

Explore the gallery of examples to get started.

INSTALLATION

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

```
pip install redisvl
```

Then make sure to have [Redis](#) accessible with Search & Query features enabled on [Redis Cloud](#) or locally in docker with [Redis Stack](#):

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

This will also spin up the [Redis Insight GUI](#) at `http://localhost:8001`.

TABLE OF CONTENTS

2.1 Overview

2.1.1 Install RedisVL

There are a few ways to install RedisVL. The easiest way is to use pip.

Install RedisVL with Pip

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

```
$ pip install -U redisvl
```

RedisVL comes with a few dependencies that are automatically installed, however, a few dependencies are optional and can be installed separately if needed:

```
$ pip install redisvl[all] # install vectorizer dependencies
$ pip install redisvl[dev] # install dev dependencies
```

If you use ZSH, remember to escape the brackets:

```
$ pip install redisvl\[all\]
```

This library supports the use of hiredis, so you can also install by running:

```
pip install redisvl[hiredis]
```

Install RedisVL from Source

To install RedisVL from source, clone the repository and install the package using pip:

```
$ git clone https://github.com/redis/redis-vl-python.git && cd redisvl
$ pip install .

# or for an editable installation (for developers of RedisVL)
$ pip install -e .
```

Installing Redis

RedisVL requires a distribution of Redis that supports the [Search and Query](#) capability of which there are 3: offering

1. [Redis Cloud](#), a fully managed cloud offering
2. [Redis Stack](#), a local docker image for testing and development
3. [Redis Enterprise](#), a commercial self-hosted

Redis Cloud

Redis Cloud is the easiest way to get started with RedisVL. You can sign up for a free account [here](#). Make sure to have the [Search](#) and [Query](#) capability enabled when creating your database.

Redis Stack (local development)

For local development and testing, Redis-Stack can be used. We recommend running Redis in a docker container. To do so, run the following command:

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

This will also spin up the [Redis Insight GUI](#) at <http://localhost:8001>.

Redis Enterprise (self-hosted)

Redis Enterprise is a commercial offering that can be self-hosted. You can download the latest version [here](#).

If you are considering a self-hosted Redis Enterprise deployment on Kubernetes, there is the [Redis Enterprise Operator](#) for Kubernetes. This will allow you to easily deploy and manage a Redis Enterprise cluster on Kubernetes.

Redis Sentinel

For high availability deployments, RedisVL supports connecting to Redis through Sentinel. Use the `redis+sentinel://` URL scheme to connect:

```
from redisvl.index import SearchIndex

# Connect via Sentinel
# Format: redis+sentinel://[username:password@]host1:port1,host2:port2/service_name[/db]
index = SearchIndex.from_yaml(
    "schema.yaml",
    redis_url="redis+sentinel://sentinel1:26379,sentinel2:26379/mymaster"
)

# With authentication
index = SearchIndex.from_yaml(
    "schema.yaml",
    redis_url="redis+sentinel://user:pass@sentinel1:26379,sentinel2:26379/mymaster/0"
)
```

The Sentinel URL format supports:

- Multiple sentinel hosts (comma-separated)
- Optional authentication (username:password)
- Service name (required - the name of the Redis master)
- Optional database number (defaults to 0)

2.1.2 The RedisVL CLI

RedisVL is a Python library with a dedicated CLI to help load and create vector search indices within Redis.

This notebook will walk through how to use the Redis Vector Library CLI (rvl).

Before running this notebook, be sure to

1. Have installed `redisvl` and have that environment active for this notebook.
2. Have a running Redis instance with the Search and Query capability

```
# First, see if the rvl tool is installed
!rvl version
```

```
11:20:38 [RedisVL] INFO   RedisVL version 0.8.2
```

Commands

Here's a table of all the rvl commands and options. We'll go into each one in detail below.

Com- mand	Options	Description
rvl version		display the redisvl library version
rvl index	create --schema or -s <schema.yaml>	create a redis index from the specified schema file
rvl index	listall	list all the existing search indices
rvl index	info --index or -i <index_name>	display the index definition in tabular format
rvl index	delete --index or -i <index_name>	remove the specified index, leaving the data still in Redis
rvl index	destroy --index or -i <index_name>	remove the specified index, as well as the associated data
rvl stats	--index or -i <index_name>	display the index statistics, including number of docs, average bytes per record, indexing time, etc
rvl stats	--schema or -s <schema.yaml>	display the index statistics of a schema defined in <schema.yaml>. The index must have already been created within Redis

Index

The `rvl index` command can be used for a number of tasks related to creating and managing indices. Whether you are working in Python or another language, this cli tool can still be useful for managing and inspecting your indices.

First, we will create an index from a yaml schema that looks like the following:

```
%%writefile schema.yaml

version: '0.1.0'

index:
  name: vectorizers
  prefix: doc
  storage_type: hash

fields:
  - name: sentence
    type: text
  - name: embedding
    type: vector
    attrs:
      dims: 768
      algorithm: flat
      distance_metric: cosine
```

Overwriting schema.yaml

```
# Create an index from a yaml schema
!rvl index create -s schema.yaml
```

```
12:42:45 [RedisVL] INFO    Index created successfully
```

```
# list the indices that are available
!rvl index listall
```

```
12:42:47 [RedisVL] INFO    Indices:
12:42:47 [RedisVL] INFO    1. vectorizers
```

```
# inspect the index fields
!rvl index info -i vectorizers
```

Index Information:

Index Name	Storage Type	Prefixes	Index Options	Indexing
vectorizers	HASH	['doc']	[]	0

Index Fields:

Name	Attribute	Type	Field Option	Option Value
↪ Field Option	Option Value	Field Option	Option Value	Field

(continues on next page)

(continued from previous page)

Option	Option Value				
→ sentence	sentence	TEXT	WEIGHT	1	→
→					→
→ embedding	embedding	VECTOR	algorithm	FLAT	→
→ data_type	FLOAT32	dim	768	distance_	
→ metric	COSINE				

```
# delete an index without deleting the data within it
!rvl index delete -i vectorizers
```

```
12:42:54 [RedisVL] INFO Index deleted successfully
```

```
# see the indices that still exist
!rvl index listall
```

```
12:42:56 [RedisVL] INFO Indices:
```

Stats

The `rvl stats` command will return some basic information about the index. This is useful for checking the status of an index, or for getting information about the index to use in other commands.

```
# create a new index with the same schema
# recreating the index will reindex the documents
!rvl index create -s schema.yaml
```

```
12:42:59 [RedisVL] INFO Index created successfully
```

```
# list the indices that are available
!rvl index listall
```

```
12:43:01 [RedisVL] INFO Indices:
12:43:01 [RedisVL] INFO 1. vectorizers
```

```
# see all the stats for the index
!rvl stats -i vectorizers
```

Statistics:

Stat Key	Value
num_docs	0
num_terms	0
max_doc_id	0
num_records	0

(continues on next page)

(continued from previous page)

percent_indexed	1
hash_indexing_failures	0
number_of_uses	1
bytes_per_record_avg	nan
doc_table_size_mb	0
inverted_sz_mb	0
key_table_size_mb	0
offset_bits_per_record_avg	nan
offset_vectors_sz_mb	0
offsets_per_term_avg	nan
records_per_doc_avg	nan
sortable_values_size_mb	0
total_indexing_time	0
total_inverted_index_blocks	0
vector_index_sz_mb	0.00818634

Optional arguments

You can modify these commands with the below optional arguments

Argument	Description	Default
-u --url	The full Redis URL to connect to	redis://localhost:6379
--host	Redis host to connect to	localhost
-p --port	Redis port to connect to. Must be an integer	6379
--user	Redis username, if one is required	default
--ssl	Boolean flag indicating if ssl is required. If set the Redis base url changes to rediss://	None
-a	Redis password, if one is required	""
--password		

Choosing your Redis instance

By default rvl first checks if you have REDIS_URL environment variable defined and tries to connect to that. If not, it then falls back to localhost:6379, unless you pass the --host or --port arguments

```
# specify your Redis instance to connect to
!rvl index listall --host localhost --port 6379
```

```
12:43:06 [RedisVL] INFO Indices:
12:43:06 [RedisVL] INFO 1. vectorizers
```

Using SSL encryption

If your Redis instance is configured to use SSL encryption then set the `--ssl` flag. You can similarly specify the username and password to construct the full Redis URL

```
# connect to rediss://jane_doe:password123@localhost:6379
!rvl index listall --user jane_doe -a password123 --ssl
```

```
!rvl index destroy -i vectorizers
```

```
12:43:09 [RedisVL] INFO    Index deleted successfully
```

2.2 RedisVL API

Reference documentation for the RedisVL API.

2.2.1 Schema

Schema in RedisVL provides a structured format to define index settings and field configurations using the following three components:

Component	Description
<i>version</i>	The version of the schema spec. Current supported version is <i>0.1.0</i> .
<i>index</i>	Index specific settings like name, key prefix, key separator, and storage type.
<i>fields</i>	Subset of fields within your data to include in the index and any custom settings.

IndexSchema

class IndexSchema(*, *index*, *fields*=<factory>, *version*='0.1.0')

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:

```
version: '0.1.0'

index:
  name: user-index
  prefix: user
  key_separator: ":"
  storage_type: json

fields:
  - name: user
    type: tag
```

(continues on next page)

(continued from previous page)

```
- 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:

```
from redisvl.schema import IndexSchema

schema = IndexSchema.from_yaml("schema.yaml")
```

Loading the schema for RedisVL from dict is as simple as:

```
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.

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **index** (*IndexInfo*)
- **fields** (*Dict[str, BaseField]*)
- **version** (*Literal['0.1.0']*)

add_field(*field_inputs*)

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.

Parameters

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.

```
# 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"
    }
})
```

add_fields(*fields*)

Extends the schema with additional fields.

This method allows dynamically adding new fields to the index schema. It processes a list of field definitions.

Parameters

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.

```
schema.add_fields([
    {"name": "user", "type": "tag"},
    {"name": "bio", "type": "text"},
    {
        "name": "user-embedding",
        "type": "vector",
        "attrs": {
            "dims": 1024,
            "algorithm": "flat",
            "datatype": "float32"
        }
    }
])
```

(continues on next page)

(continued from previous page)

```

    }
  }
])

```

classmethod `from_dict(data)`

Create an IndexSchema from a dictionary.

Parameters

data (*Dict[str, Any]*) – The index schema data.

Returns

The index schema.

Return type

IndexSchema

```

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
            }
        }
    ]
})

```

classmethod `from_yaml(file_path)`

Create an IndexSchema from a YAML file.

Parameters

file_path (*str*) – The path to the YAML file.

Returns

The index schema.

Return type

IndexSchema

```

from redisvl.schema import IndexSchema

schema = IndexSchema.from_yaml("schema.yaml")

```

remove_field(*field_name*)

Removes a field from the schema based on the specified name.

This method is useful for dynamically altering the schema by removing existing fields.

Parameters

field_name (*str*) – The name of the field to be removed.

to_dict()

Serialize the index schema model to a dictionary, handling Enums and other special cases properly.

Returns

The index schema as a dictionary.

Return type

Dict[str, Any]

to_yaml(*file_path*, *overwrite=True*)

Write the index schema to a YAML file.

Parameters

- **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.

Return type

None

property field_names: List[str]

A list of field names associated with the index schema.

Returns

A list of field names from the schema.

Return type

List[str]

fields: Dict[str, BaseField]

Fields associated with the search index and their properties.

Note: When creating from dict/YAML, provide fields as a list of field definitions. The validator will convert them to a Dict[str, BaseField] internally.

index: IndexInfo

Details of the basic index configurations.

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to [ConfigDict][pydantic.config.ConfigDict].

version: Literal['0.1.0']

Version of the underlying index schema.

Defining Fields

Fields in the schema can be defined in YAML format or as a Python dictionary, specifying a name, type, an optional path, and attributes for customization.

YAML Example:

```
- name: title
  type: text
  path: $.document.title
  attrs:
    weight: 1.0
    no_stem: false
    withsuffixtrie: true
```

Python Dictionary Example:

```
{
  "name": "location",
  "type": "geo",
  "attrs": {
    "sortable": true
  }
}
```

Basic Field Types

RedisVL supports several basic field types for indexing different kinds of data. Each field type has specific attributes that customize its indexing and search behavior.

Text Fields

Text fields support full-text search with stemming, phonetic matching, and other text analysis features.

class `TextField`(*, name, type=`FieldTypes.TEXT`, path=`None`, attrs=<factory>)

Bases: `BaseField`

Text field supporting a full text search index

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **name** (*str*)
- **type** (*Literal[FieldTypes.TEXT]*)
- **path** (*str | None*)
- **attrs** (*TextFieldAttributes*)

as_redis_field()

Convert schema field to Redis Field object

Return type

Field

attrs: *TextFieldAttributes*

Specified field attributes

model_config: `ClassVar[ConfigDict] = {}`

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

type: `Literal[FieldTypes.TEXT]`

Field type

```
class TextFieldAttributes(*, sortable=False, index_missing=False, no_index=False, weight=1,
                          no_stem=False, withsuffixtrie=False, phonetic_matcher=None,
                          index_empty=False, unf=False)
```

Full text field attributes

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **sortable** (*bool*)
- **index_missing** (*bool*)
- **no_index** (*bool*)
- **weight** (*float*)
- **no_stem** (*bool*)
- **withsuffixtrie** (*bool*)
- **phonetic_matcher** (*str | None*)
- **index_empty** (*bool*)
- **unf** (*bool*)

index_empty: `bool`

Allow indexing and searching for empty strings

model_config: `ClassVar[ConfigDict] = {}`

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

no_stem: `bool`

Disable stemming on the text field during indexing

phonetic_matcher: `str | None`

Used to perform phonetic matching during search

unf: `bool`

Un-normalized form - disable normalization on sortable fields (only applies when `sortable=True`)

weight: float

Declares the importance of this field when calculating results

withsuffixtrie: bool

Keep a suffix trie with all terms which match the suffix to optimize certain queries

Tag Fields

Tag fields are optimized for exact-match filtering and faceted search on categorical data.

class TagField(**, name, type=FieldTypes.TAG, path=None, attrs=<factory>*)

Bases: BaseField

Tag field for simple boolean-style filtering

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **name** (*str*)
- **type** (*Literal[FieldTypes.TAG]*)
- **path** (*str | None*)
- **attrs** (*TagFieldAttributes*)

as_redis_field()

Convert schema field to Redis Field object

Return type

Field

attrs: *TagFieldAttributes*

Specified field attributes

model_config: *ClassVar[ConfigDict] = {}*

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

type: *Literal[FieldTypes.TAG]*

Field type

class TagFieldAttributes(**, sortable=False, index_missing=False, no_index=False, separator=',', case_sensitive=False, withsuffixtrie=False, index_empty=False*)

Tag field attributes

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **sortable** (*bool*)

- **index_missing** (*bool*)
- **no_index** (*bool*)
- **separator** (*str*)
- **case_sensitive** (*bool*)
- **withsuffixtrie** (*bool*)
- **index_empty** (*bool*)

case_sensitive: **bool**

Treat text as case sensitive or not. By default, tag characters are converted to lowercase

index_empty: **bool**

Allow indexing and searching for empty strings

model_config: **ClassVar**[**ConfigDict**] = {}

Configuration for the model, should be a dictionary conforming to *[ConfigDict][pydantic.config.ConfigDict]*.

separator: **str**

Indicates how the text in the original attribute is split into individual tags

withsuffixtrie: **bool**

Keep a suffix trie with all terms which match the suffix to optimize certain queries

Numeric Fields

Numeric fields support range queries and sorting on numeric data.

class **NumericField**(**, name, type=FieldTypes.NUMERIC, path=None, attrs=<factory>*)

Bases: **BaseField**

Numeric field for numeric range filtering

Create a new model by parsing and validating input data from keyword arguments.

Raises *[ValidationError][pydantic_core.ValidationError]* if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **name** (*str*)
- **type** (*Literal[FieldTypes.NUMERIC]*)
- **path** (*str | None*)
- **attrs** (*NumericFieldAttributes*)

as_redis_field()

Convert schema field to Redis Field object

Return type

Field

attrs: *NumericFieldAttributes*

Specified field attributes

```
model_config: ClassVar[ConfigDict] = {}
```

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

```
type: Literal[FieldTypes.NUMERIC]
```

Field type

```
class NumericFieldAttributes(*, sortable=False, index_missing=False, no_index=False, unf=False)
```

Numeric field attributes

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **sortable** (*bool*)
- **index_missing** (*bool*)
- **no_index** (*bool*)
- **unf** (*bool*)

```
model_config: ClassVar[ConfigDict] = {}
```

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

```
unf: bool
```

Un-normalized form - disable normalization on sortable fields (only applies when `sortable=True`)

Geo Fields

Geo fields enable location-based search with geographic coordinates.

```
class GeoField(*, name, type=FieldTypes.GEO, path=None, attrs=<factory>)
```

Bases: `BaseField`

Geo field with a geo-spatial index for location based search

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **name** (*str*)
- **type** (`Literal[FieldTypes.GEO]`)
- **path** (*str* | *None*)
- **attrs** (`GeoFieldAttributes`)

as_redis_field()

Convert schema field to Redis Field object

Return type

Field

attrs: *GeoFieldAttributes*

Specified field attributes

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to *[ConfigDict][pydantic.config.ConfigDict]*.

type: Literal[FieldTypes.GEO]

Field type

class GeoFieldAttributes(*, sortable=False, index_missing=False, no_index=False)

Numeric field attributes

Create a new model by parsing and validating input data from keyword arguments.

Raises *[ValidationError][pydantic_core.ValidationError]* if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **sortable** (*bool*)
- **index_missing** (*bool*)
- **no_index** (*bool*)

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to *[ConfigDict][pydantic.config.ConfigDict]*.

Vector Field Types

Vector fields enable semantic similarity search using various algorithms. All vector fields share common attributes but have algorithm-specific configurations.

Common Vector Attributes

All vector field types share these base attributes:

class BaseVectorFieldAttributes(*, dims, algorithm, datatype=VectorDataType.FLOAT32, distance_metric=VectorDistanceMetric.COSINE, initial_cap=None, index_missing=False)

Base vector field attributes shared by FLAT, HNSW, and SVS-VAMANA fields

Create a new model by parsing and validating input data from keyword arguments.

Raises *[ValidationError][pydantic_core.ValidationError]* if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **dims** (*int*)
- **algorithm** (*VectorIndexAlgorithm*)
- **datatype** (*VectorDataType*)
- **distance_metric** (*VectorDistanceMetric*)
- **initial_cap** (*int* | *None*)
- **index_missing** (*bool*)

classmethod uppercase_strings(*v*)

Validate that provided values are cast to uppercase

algorithm: **VectorIndexAlgorithm**

FLAT, HNSW, or SVS-VAMANA

Type

The indexing algorithm for the field

datatype: **VectorDataType**

The float datatype for the vector embeddings

dims: **int**

Dimensionality of the vector embeddings field

distance_metric: **VectorDistanceMetric**

The distance metric used to measure query relevance

property field_data: **Dict[str, Any]**

Select attributes required by the Redis API

index_missing: **bool**

Allow indexing and searching for missing values (documents without the field)

initial_cap: **int** | **None**

Initial vector capacity in the index affecting memory allocation size of the index

model_config: **ClassVar[ConfigDict]** = {}

Configuration for the model, should be a dictionary conforming to *[ConfigDict][pydantic.config.ConfigDict]*.

Key Attributes:

- *dims*: Dimensionality of the vector (e.g., 768, 1536).
- *algorithm*: Indexing algorithm for vector search:
 - *flat*: Brute-force exact search. 100% recall, slower for large datasets. Best for <10K vectors.
 - *hns*: Graph-based approximate search. Fast with high recall (95-99%). Best for general use.
 - *svs-vamana*: SVS-VAMANA (Scalable Vector Search with VAMANA graph algorithm) provides fast approximate nearest neighbor search with optional compression support. This algorithm is optimized for Intel hardware and offers reduced memory usage through vector compression.

Note

For detailed algorithm comparison and selection guidance, see *Vector Algorithm Comparison*.

- *datatype*: Float precision (*bfloat16*, *float16*, *float32*, *float64*). Note: SVS-VAMANA only supports *float16* and *float32*.
- *distance_metric*: Similarity metric (*COSINE*, *L2*, *IP*).
- *initial_cap*: Initial capacity hint for memory allocation (optional).
- *index_missing*: When True, allows searching for documents missing this field (optional).

HNSW Vector Fields

HNSW (Hierarchical Navigable Small World) - Graph-based approximate search with excellent recall. **Best for general-purpose vector search (10K-1M+ vectors).**

When to use HNSW & Performance Details

Use HNSW when:

- Medium to large datasets (100K-1M+ vectors) requiring high recall rates
- Search accuracy is more important than memory usage
- Need general-purpose vector search with balanced performance
- Cross-platform deployments where hardware-specific optimizations aren't available

Performance characteristics:

- **Search speed**: Very fast approximate search with tunable accuracy
- **Memory usage**: Higher than compressed SVS-VAMANA but reasonable for most applications
- **Recall quality**: Excellent recall rates (95-99%), often better than other approximate methods
- **Build time**: Moderate construction time, faster than SVS-VAMANA for smaller datasets

```
class HNSWVectorField(*, name, type='vector', path=None, attrs)
```

Bases: BaseField

Vector field with HNSW (Hierarchical Navigable Small World) indexing for approximate nearest neighbor search.

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **name** (*str*)
- **type** (*Literal*['vector'])
- **path** (*str* | *None*)
- **attrs** (*HNSWVectorFieldAttributes*)

```
as_redis_field()
```

Convert schema field to Redis Field object

Return type

Field

attrs: *HNSWVectorFieldAttributes*

Specified field attributes

model_config: `ClassVar[ConfigDict] = {}`

Configuration for the model, should be a dictionary conforming to *[ConfigDict][pydantic.config.ConfigDict]*.

type: `Literal['vector']`

Field type

```
class HNSWVectorFieldAttributes(*, dims, algorithm=VectorIndexAlgorithm.HNSW,
                                datatype=VectorDataType.FLOAT32,
                                distance_metric=VectorDistanceMetric.COSINE, initial_cap=None,
                                index_missing=False, m=16, ef_construction=200, ef_runtime=10,
                                epsilon=0.01)
```

HNSW vector field attributes for approximate nearest neighbor search.

Create a new model by parsing and validating input data from keyword arguments.

Raises *[ValidationError][pydantic_core.ValidationError]* if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **dims** (*int*)
- **algorithm** (*Literal[VectorIndexAlgorithm.HNSW]*)
- **datatype** (*VectorDataType*)
- **distance_metric** (*VectorDistanceMetric*)
- **initial_cap** (*int | None*)
- **index_missing** (*bool*)
- **m** (*int*)
- **ef_construction** (*int*)
- **ef_runtime** (*int*)
- **epsilon** (*float*)

algorithm: `Literal[VectorIndexAlgorithm.HNSW]`

The indexing algorithm (fixed as 'hnsw')

ef_construction: `int`

100-800)

Type

Max edge candidates during build time (default

Type

200, range

ef_runtime: `int`

- 10) • primary tuning parameter

Type

Max top candidates during search (default

epsilon: float

0.01)

Type

Range search boundary factor (default

m: int

8-64)

Type

Max outgoing edges per node in each layer (default

Type

16, range

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to [ConfigDict][pydantic.config.ConfigDict].

HNSW Examples:

Balanced configuration (recommended starting point):

```
- name: embedding
  type: vector
  attrs:
    algorithm: hnsw
    dims: 768
    distance_metric: cosine
    datatype: float32
    # Balanced settings for good recall and performance
    m: 16
    ef_construction: 200
    ef_runtime: 10
```

High-recall configuration:

```
- name: embedding
  type: vector
  attrs:
    algorithm: hnsw
    dims: 768
    distance_metric: cosine
    datatype: float32
    # Tuned for maximum accuracy
    m: 32
    ef_construction: 400
    ef_runtime: 50
```

SVS-VAMANA Vector Fields

SVS-VAMANA (Scalable Vector Search with VAMANA graph algorithm) provides fast approximate nearest neighbor search with optional compression support. This algorithm is optimized for Intel hardware and offers reduced memory usage through vector compression. **Best for large datasets (>100K vectors) on Intel hardware with memory constraints.**

When to use SVS-VAMANA & Detailed Guide

Requirements:

- Redis \geq 8.2.0 with RediSearch \geq 2.8.10
- datatype must be 'float16' or 'float32' (float64/bfloat16 not supported)

Use SVS-VAMANA when:

- Large datasets where memory is expensive
- Cloud deployments with memory-based pricing
- When 90-95% recall is acceptable
- High-dimensional vectors (>1024 dims) with LeanVec compression

Performance vs other algorithms:

- **vs FLAT:** Much faster search, significantly lower memory usage with compression, but approximate results
- **vs HNSW:** Better memory efficiency with compression, similar or better recall, Intel-optimized

Compression selection guide:

- **No compression:** Best performance, standard memory usage
- **LVQ4/LVQ8:** Good balance of compression (2x-4x) and performance
- **LeanVec4x8/LeanVec8x8:** Maximum compression (up to 8x) with dimensionality reduction

Memory Savings Examples (1M vectors, 768 dims):

- No compression (float32): 3.1 GB
- LVQ4x4 compression: 1.6 GB (~48% savings)
- LeanVec4x8 + reduce to 384: 580 MB (~81% savings)

class SVSVectorField(*, name, type=*FieldTypes.VECTOR*, path=None, attrs)

Bases: BaseField

Vector field with SVS-VAMANA indexing and compression for memory-efficient approximate nearest neighbor search.

Create a new model by parsing and validating input data from keyword arguments.

Raises [*ValidationError*][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **name** (*str*)
- **type** (*Literal*[*FieldTypes.VECTOR*])

- **path** (*str* | *None*)
- **attrs** (*SVSVectorFieldAttributes*)

as_redis_field()

Convert schema field to Redis Field object

Return type

Field

attrs: *SVSVectorFieldAttributes*

Specified field attributes

model_config: *ClassVar[ConfigDict]* = {}

Configuration for the model, should be a dictionary conforming to *[ConfigDict][pydantic.config.ConfigDict]*.

type: *Literal[FieldTypes.VECTOR]*

Field type

```
class SVSVectorFieldAttributes(*, dims, algorithm=VectorIndexAlgorithm.SVS_VAMANA,
                             datatype=VectorDataType.FLOAT32,
                             distance_metric=VectorDistanceMetric.COSINE, initial_cap=None,
                             index_missing=False, graph_max_degree=40,
                             construction_window_size=250, search_window_size=20, epsilon=0.01,
                             compression=None, reduce=None, training_threshold=None)
```

SVS-VAMANA vector field attributes with compression support.

Create a new model by parsing and validating input data from keyword arguments.

Raises *[ValidationError][pydantic_core.ValidationError]* if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **dims** (*int*)
- **algorithm** (*Literal[VectorIndexAlgorithm.SVS_VAMANA]*)
- **datatype** (*VectorDataType*)
- **distance_metric** (*VectorDistanceMetric*)
- **initial_cap** (*int* | *None*)
- **index_missing** (*bool*)
- **graph_max_degree** (*int*)
- **construction_window_size** (*int*)
- **search_window_size** (*int*)
- **epsilon** (*float*)
- **compression** (*CompressionType* | *None*)
- **reduce** (*int* | *None*)
- **training_threshold** (*int* | *None*)

validate_svs_params()

Validate SVS-VAMANA specific constraints

algorithm: Literal[VectorIndexAlgorithm.SVS_VAMANA]

The indexing algorithm for the vector field

compression: CompressionType | None

LVQ4, LVQ8, LeanVec4x8, LeanVec8x8

Type

Vector compression

construction_window_size: int

250) • affects quality vs build time

Type

Build-time candidates (default

epsilon: float

0.01)

Type

Range query boundary factor (default

graph_max_degree: int

40) • affects recall vs memory

Type

Max edges per node (default

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to *[ConfigDict][pydantic.config.ConfigDict]*.

reduce: int | None

Dimensionality reduction for LeanVec types (must be < dims)

search_window_size: int

20) • primary tuning parameter

Type

Search candidates (default

training_threshold: int | None

10,240)

Type

Min vectors before compression training (default

SVS-VAMANA Examples:

Basic configuration (no compression):

```
- name: embedding
  type: vector
  attrs:
    algorithm: svsvamana
    dims: 768
    distance_metric: cosine
    datatype: float32
    # Standard settings for balanced performance
    graph_max_degree: 40
    construction_window_size: 250
    search_window_size: 20
```

High-performance configuration with compression:

```
- name: embedding
  type: vector
  attrs:
    algorithm: svsvamana
    dims: 768
    distance_metric: cosine
    datatype: float32
    # Tuned for better recall
    graph_max_degree: 64
    construction_window_size: 500
    search_window_size: 40
    # Maximum compression with dimensionality reduction
    compression: LeanVec4x8
    reduce: 384 # 50% dimensionality reduction
    training_threshold: 1000
```

Important Notes:

- **Requirements:** SVS-VAMANA requires Redis ≥ 8.2 with RediSearch $\geq 2.8.10$.
- **Datatype limitations:** SVS-VAMANA only supports *float16* and *float32* datatypes (not *bfloat16* or *float64*).
- **Compression compatibility:** The *reduce* parameter is only valid with LeanVec compression types (*LeanVec4x8* or *LeanVec8x8*).
- **Platform considerations:** Intel's proprietary LVQ and LeanVec optimizations are not available in Redis Open Source. On non-Intel platforms and Redis Open Source, SVS-VAMANA with compression falls back to basic 8-bit scalar quantization.
- **Performance tip:** Start with default parameters and tune *search_window_size* first for your speed vs accuracy requirements.

FLAT Vector Fields

FLAT - Brute-force exact search. **Best for small datasets (<10K vectors) requiring 100% accuracy.**

When to use FLAT & Performance Details

Use FLAT when:

- Small datasets (<100K vectors) where exact results are required
- Search accuracy is critical and approximate results are not acceptable
- Baseline comparisons when evaluating approximate algorithms
- Simple use cases where setup simplicity is more important than performance

Performance characteristics:

- **Search accuracy:** 100% exact results (no approximation)
- **Search speed:** Linear time $O(n)$ - slower as dataset grows
- **Memory usage:** Minimal overhead, stores vectors as-is
- **Build time:** Fastest index construction (no preprocessing)

Trade-offs vs other algorithms:

- **vs HNSW:** Much slower search but exact results, faster index building
- **vs SVS-VAMANA:** Slower search and higher memory usage, but exact results

class FlatVectorField(*, name, type=FieldTypes.VECTOR, path=None, attrs)

Bases: BaseField

Vector field with FLAT (exact search) indexing for exact nearest neighbor search.

Create a new model by parsing and validating input data from keyword arguments.

Raises [ValidationError][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **name** (*str*)
- **type** (*Literal[FieldTypes.VECTOR]*)
- **path** (*str | None*)
- **attrs** (*FlatVectorFieldAttributes*)

as_redis_field()

Convert schema field to Redis Field object

Return type

Field

attrs: *FlatVectorFieldAttributes*

Specified field attributes

```
model_config: ClassVar[ConfigDict] = {}
```

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

```
type: Literal[FieldTypes.VECTOR]
```

Field type

```
class FlatVectorFieldAttributes(*, dims, algorithm=VectorIndexAlgorithm.FLAT,
                                datatype=VectorDataType.FLOAT32,
                                distance_metric=VectorDistanceMetric.COSINE, initial_cap=None,
                                index_missing=False, block_size=None)
```

FLAT vector field attributes for exact nearest neighbor search.

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **dims** (*int*)
- **algorithm** (*Literal[VectorIndexAlgorithm.FLAT]*)
- **datatype** (*VectorDataType*)
- **distance_metric** (*VectorDistanceMetric*)
- **initial_cap** (*int | None*)
- **index_missing** (*bool*)
- **block_size** (*int | None*)

```
algorithm: Literal[VectorIndexAlgorithm.FLAT]
```

The indexing algorithm (fixed as 'flat')

```
block_size: int | None
```

Block size for processing (optional) - improves batch operation throughput

```
model_config: ClassVar[ConfigDict] = {}
```

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

FLAT Example:

```
- name: embedding
  type: vector
  attrs:
    algorithm: flat
    dims: 768
    distance_metric: cosine
    datatype: float32
    # Optional: tune for batch processing
    block_size: 1024
```

Note: FLAT is recommended for small datasets or when exact results are mandatory. For larger datasets, consider HNSW or SVS-VAMANA for better performance.

SVS-VAMANA Configuration Utilities

For SVS-VAMANA indices, RedisVL provides utilities to help configure compression settings and estimate memory savings.

CompressionAdvisor

class CompressionAdvisor

Bases: object

Helper to recommend compression settings based on vector characteristics.

This class provides utilities to: - Recommend optimal SVS-VAMANA configurations based on vector dimensions and priorities - Estimate memory savings from compression and dimensionality reduction

Examples

```
>>> # Get recommendations for high-dimensional vectors
>>> config = CompressionAdvisor.recommend(dims=1536, priority="balanced")
>>> config.compression
'LeanVec4x8'
>>> config.reduce
768
```

```
>>> # Estimate memory savings
>>> savings = CompressionAdvisor.estimate_memory_savings(
...     compression="LeanVec4x8",
...     dims=1536,
...     reduce=768
... )
>>> savings
81.2
```

static estimate_memory_savings(*compression, dims, reduce=None*)

Estimate memory savings percentage from compression.

Calculates the percentage of memory saved compared to uncompressed float32 vectors.

Parameters

- **compression** (*str*) – Compression type (e.g., “LVQ4”, “LeanVec4x8”)
- **dims** (*int*) – Original vector dimensionality
- **reduce** (*int* | *None*) – Reduced dimensionality (for LeanVec compression)

Returns

Memory savings percentage (0-100)

Return type

float

Examples

```
>>> # LeanVec with dimensionality reduction
>>> CompressionAdvisor.estimate_memory_savings(
...     compression="LeanVec4x8",
...     dims=1536,
...     reduce=768
... )
81.2
```

```
>>> # LVQ without dimensionality reduction
>>> CompressionAdvisor.estimate_memory_savings(
...     compression="LVQ4",
...     dims=384
... )
87.5
```

static recommend(*dims*, *priority*='balanced', *datatype*=None)

Recommend compression settings based on dimensions and priorities.

Parameters

- **dims** (*int*) – Vector dimensionality (must be > 0)
- **priority** (*Literal['speed', 'memory', 'balanced']*) – Optimization priority: - “memory”: Maximize memory savings - “speed”: Optimize for query speed - “balanced”: Balance between memory and speed
- **datatype** (*str* / *None*) – Override datatype (default: float16 for high-dim, float32 for low-dim)

Returns

Complete SVS-VAMANA configuration including:

- algorithm: “svs-vamana”
- datatype: Recommended datatype
- compression: Compression type
- reduce: Dimensionality reduction (for LeanVec only)
- graph_max_degree: Graph connectivity
- construction_window_size: Build-time candidates
- search_window_size: Query-time candidates

Return type

dict

Raises

ValueError – If *dims* <= 0

Examples

```
>>> # High-dimensional embeddings (e.g., OpenAI ada-002)
>>> config = CompressionAdvisor.recommend(dims=1536, priority="memory")
>>> config.compression
'LeanVec4x8'
>>> config.reduce
768
```

```
>>> # Lower-dimensional embeddings
>>> config = CompressionAdvisor.recommend(dims=384, priority="speed")
>>> config.compression
'LVQ4x8'
```

SVSConfig

```
class SVSConfig(*, algorithm='svs-vamana', datatype=None, compression=None, reduce=None,
                graph_max_degree=None, construction_window_size=None, search_window_size=None)
```

Bases: BaseModel

SVS-VAMANA configuration model.

Parameters

- **algorithm** (*Literal*['svs-vamana'])
- **datatype** (*str* | *None*)
- **compression** (*str* | *None*)
- **reduce** (*int* | *None*)
- **graph_max_degree** (*int* | *None*)
- **construction_window_size** (*int* | *None*)
- **search_window_size** (*int* | *None*)

algorithm

Always “svs-vamana”

Type

Literal['svs-vamana']

datatype

Vector datatype (float16, float32)

Type

str | *None*

compression

Compression type (LVQ4, LeanVec4x8, etc.)

Type

str | *None*

reduce

Reduced dimensionality (only for LeanVec)

Type

int | None

graph_max_degree

Max edges per node

Type

int | None

construction_window_size

Build-time candidates

Type

int | None

search_window_size

Query-time candidates

Type

int | None

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

Vector Algorithm Comparison

This section provides detailed guidance for choosing between vector search algorithms.

Algorithm Selection Guide

Table 1: Vector Algorithm Comparison

Algorithm	Best For	Performance	Memory Usage	Trade-offs
FLAT	Small datasets (<100K vectors)	100% recall, O(n) search	Minimal overhead	Exact but slow for large data
HNSW	General purpose (100K-1M+ vectors)	95-99% recall, O(log n) search	Moderate (graph overhead)	Fast approximate search
SVS-VAMANA	Large datasets with memory constraints	90-95% recall, O(log n) search	Low (with compression)	Intel-optimized, requires newer Redis

When to Use Each Algorithm

Choose FLAT when:

- Dataset size < 100,000 vectors
- Exact results are mandatory
- Simple setup is preferred
- Query latency is not critical

Choose HNSW when:

- Dataset size 100K - 1M+ vectors
- Need balanced speed and accuracy
- Cross-platform compatibility required
- Most common choice for production

Choose SVS-VAMANA when:

- Dataset size > 100K vectors
- Memory usage is a primary concern
- Running on Intel hardware
- Can accept 90-95% recall for memory savings

Performance Characteristics

Search Speed:

- FLAT: Linear time $O(n)$ - gets slower as data grows
- HNSW: Logarithmic time $O(\log n)$ - scales well
- SVS-VAMANA: Logarithmic time $O(\log n)$ - scales well

Memory Usage (1M vectors, 768 dims, float32):

- FLAT: ~3.1 GB (baseline)
- HNSW: ~3.7 GB (20% overhead for graph)
- SVS-VAMANA: 1.6-3.1 GB (depends on compression)

Recall Quality:

- FLAT: 100% (exact search)
- HNSW: 95-99% (tunable via `ef_runtime`)
- SVS-VAMANA: 90-95% (depends on compression)

Migration Considerations

From FLAT to HNSW:

- Straightforward migration
- Expect slight recall reduction but major speed improvement
- Tune `ef_runtime` to balance speed vs accuracy

From HNSW to SVS-VAMANA:

- Requires Redis ≥ 8.2 with RediSearch $\geq 2.8.10$
- Change datatype to float16 or float32 if using others
- Consider compression options for memory savings

From SVS-VAMANA to others:

- May need to change datatype back if using float64/bfloat16
- HNSW provides similar performance with broader compatibility

For complete Redis field documentation, see: <https://redis.io/commands/ft.create/>

2.2.2 Search Index Classes

Class	Description
<i>SearchIndex</i>	Primary class to write, read, and search across data structures in Redis.
<i>AsyncSearchIndex</i>	Async version of the SearchIndex to write, read, and search across data structures in Redis.

SearchIndex

```
class SearchIndex(schema, redis_client=None, redis_url=None, connection_kwargs=None,
                  validate_on_load=False, **kwargs)
```

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.

```
from redisvl.index import SearchIndex

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

# create the index
index.create(overwrite=True, drop=False)

# data is an iterable of dictionaries
```

(continues on next page)

(continued from previous page)

```
index.load(data)

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

Initialize the RedisVL search index with a schema, Redis client (or URL string with other connection args), connection_args, and other kwargs.

Parameters

- **schema** (*IndexSchema*) – Index schema object.
- **redis_client** (*Optional[Redis]*) – An instantiated redis client.
- **redis_url** (*Optional[str]*) – The URL of the Redis server to connect to.
- **connection_kwargs** (*Dict[str, Any], optional*) – Redis client connection args.
- **validate_on_load** (*bool, optional*) – Whether to validate data against schema when loading. Defaults to False.

aggregate(*args, **kwargs)

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

Raw Redis aggregation results.

Return type

Result

batch_query(queries, batch_size=10)

Execute a batch of queries and process results.

Parameters

- **queries** (*Sequence[BaseQuery]*)
- **batch_size** (*int*)

Return type

List[List[Dict[str, Any]]]

batch_search(queries, batch_size=10)

Perform a search against the index for multiple queries.

This method takes a list of queries and optionally query params and returns a list of Result objects for each query. Results are returned in the same order as the queries.

NOTE: Cluster users may need to incorporate hash tags into their query to avoid cross-slot operations.

Parameters

- **queries** (*List[SearchParams]*) – The queries to search for.
- **batch_size** (*int, optional*) – The number of queries to search for at a time. Defaults to 10.

Returns

The search results for each query.

Return type

List[Result]

clear()

Clear all keys in Redis associated with the index, leaving the index available and in-place for future insertions or updates.

NOTE: This method requires custom behavior for Redis Cluster because here, we can't easily give control of the keys we're clearing to the user so they can separate them based on hash tag.

Returns

Count of records deleted from Redis.

Return type

int

connect(*redis_url=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.

Parameters

redis_url (*Optional[str], optional*) – The URL of the Redis server to connect to.

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.
- **ModuleNotFoundError** – If required Redis modules are not installed.

create(*overwrite=False, drop=False*)

Create an index in Redis with the current schema and properties.

Parameters

- **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.

Return type

None

```
# 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)
```

delete(drop=True)

Delete the search index while optionally dropping all keys associated with the index.

Parameters

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

Raises

redis.exceptions.ResponseError – If the index does not exist.

disconnect()

Disconnect from the Redis database.

drop_documents(ids)

Remove documents from the index by their document IDs.

This method converts document IDs to Redis keys automatically by applying the index's key prefix and separator configuration.

NOTE: Cluster users will need to incorporate hash tags into their document IDs and only call this method with documents from a single hash tag at a time.

Parameters

ids (*Union[str, List[str]]*) – The document ID or IDs to remove from the index.

Returns

Count of documents deleted from Redis.

Return type

int

drop_keys(keys)

Remove a specific entry or entries from the index by it's key ID.

Parameters

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

Returns

Count of records deleted from Redis.

Return type

int

exists()

Check if the index exists in Redis.

Returns

True if the index exists, False otherwise.

Return type

bool

expire_keys(keys, ttl)

Set the expiration time for a specific entry or entries in Redis.

Parameters

- **keys** (*Union[str, List[str]]*) – The entry ID or IDs to set the expiration for.
- **ttl** (*int*) – The time-to-live in seconds.

Return type

int | List[int]

fetch(*id*)

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).

Parameters

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

Returns

The fetched object.

Return type

Dict[str, Any]

classmethod from_dict(*schema_dict*, ***kwargs*)

Create a SearchIndex from a dictionary.

Parameters

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

Returns

A RedisVL SearchIndex object.

Return type

SearchIndex

```
from redisvl.index import SearchIndex

index = SearchIndex.from_dict({
    "index": {
        "name": "my-index",
        "prefix": "rvl",
        "storage_type": "hash",
    },
    "fields": [
        {"name": "doc-id", "type": "tag"}
    ]
}, redis_url="redis://localhost:6379")
```

classmethod from_existing(*name*, *redis_client=None*, *redis_url=None*, ***kwargs*)

Initialize from an existing search index in Redis by index name.

Parameters

- **name** (*str*) – Name of the search index in Redis.
- **redis_client** (*Optional[Redis]*) – An instantiated redis client.
- **redis_url** (*Optional[str]*) – The URL of the Redis server to connect to.

Raises

ValueError – If *redis_url* or *redis_client* is not provided.

classmethod from_yaml(*schema_path*, ***kwargs*)

Create a SearchIndex from a YAML schema file.

Parameters

schema_path (*str*) – Path to the YAML schema file.

Returns

A RedisVL SearchIndex object.

Return type

SearchIndex

```
from redisvl.index import SearchIndex

index = SearchIndex.from_yaml("schemas/schema.yaml", redis_url="redis://
↪localhost:6379")
```

info(*name=None*)

Get information about the index.

Parameters

name (*str*, *optional*) – Index name to fetch info about. Defaults to None.

Returns

A dictionary containing the information about the index.

Return type

dict

key(*id*)

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).

Parameters

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

Returns

The full Redis key including key prefix and value as a string.

Return type

str

listall()

List all search indices in Redis database.

Returns

The list of indices in the database.

Return type

List[str]

load(*data*, *id_field=None*, *keys=None*, *ttr=None*, *preprocess=None*, *batch_size=None*)

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.

Parameters

- **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 of keys loaded to Redis.

Return type

List[str]

Raises

- **SchemaValidationError** – If validation fails when `validate_on_load` is enabled.
- **RedisVLError** – If there's an error loading data to Redis.

paginate(*query, page_size=30*)

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.

Parameters

- **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.

Return type

Generator

```
# 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.

query(*query*)

Execute a query on the index.

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

Parameters

query (*Union[BaseQuery, AggregateQuery]*) – The query to run.

Returns

A list of search results.

Return type

List[Result]

```
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)
```

search(*args, **kwargs)

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

Raw Redis search results.

Return type

Result

set_client(redis_client, **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.

Parameters

redis_client (*Redis*) – A Redis or Async Redis client instance to be used for the connection.

Raises

TypeError – If the provided client is not valid.

property client: Redis | RedisCluster | None

The underlying redis-py client object.

property key_separator: str

The optional separator between a defined prefix and key value in forming a Redis key.

property name: str

The name of the Redis search index.

property prefix: str

The optional key prefix that comes before a unique key value in forming a Redis key. If multiple prefixes are configured, returns the first one.

property storage_type: StorageType

The underlying storage type for the search index; either hash or json.

AsyncSearchIndex

class AsyncSearchIndex(*schema*, *, *redis_url=None*, *redis_client=None*, *connection_kwargs=None*, *validate_on_load=False*, ***kwargs*)

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.

```
from redisvl.index import AsyncSearchIndex

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

# create the index
await index.create(overwrite=True, drop=False)

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

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

Initialize the RedisVL async search index with a schema.

Parameters

- **schema** ([IndexSchema](#)) – Index schema object.
- **redis_url** (*Optional[str]*, *optional*) – The URL of the Redis server to connect to.
- **redis_client** (*Optional[AsyncRedis]*) – An instantiated redis client.
- **connection_kwargs** (*Optional[Dict[str, Any]]*) – Redis client connection args.
- **validate_on_load** (*bool*, *optional*) – Whether to validate data against schema when loading. Defaults to False.

async aggregate(**args*, ***kwargs*)

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

Raw Redis aggregation results.

Return type

Result

async batch_query(*queries*, *batch_size=10*)

Asynchronously execute a batch of queries and process results.

Parameters

- **queries** (*List[BaseQuery]*)

- **batch_size** (*int*)

Return type

List[List[Dict[str, Any]]]

async batch_search(*queries, batch_size=10*)

Asynchronously execute a batch of search queries.

This method takes a list of search queries and executes them in batches to improve performance when dealing with multiple queries.

NOTE: Cluster users may need to incorporate hash tags into their query to avoid cross-slot operations.

Parameters

- **queries** (*List[SearchParams]*) – A list of search queries to execute. Each query can be either a string or a tuple of (query, params).
- **batch_size** (*int, optional*) – The number of queries to execute in each batch. Defaults to 10.

Returns

A list of search results corresponding to each query.

Return type

List[Result]

```
queries = [  
    "hello world",  
    ("goodbye world", {"num_results": 5}),  
]  
  
results = await index.batch_search(queries)
```

async clear()

Clear all keys in Redis associated with the index, leaving the index available and in-place for future insertions or updates.

NOTE: This method requires custom behavior for Redis Cluster because here, we can't easily give control of the keys we're clearing to the user so they can separate them based on hash tag.

Returns

Count of records deleted from Redis.

Return type

int

connect(*redis_url=None, **kwargs*)

[DEPRECATED] Connect to a Redis instance. Use connection parameters in `__init__`.

Parameters

redis_url (*str | None*)

async create(*overwrite=False, drop=False*)

Asynchronously create an index in Redis with the current schema and properties.

Parameters

- **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.

Return type

None

```
# 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)
```

async delete(drop=True)

Delete the search index.

Parameters**drop** (*bool*, *optional*) – Delete the documents in the index. Defaults to True.**Raises****redis.exceptions.ResponseError** – If the index does not exist.**async disconnect()**

Disconnect from the Redis database.

async drop_documents(ids)

Remove documents from the index by their document IDs.

This method converts document IDs to Redis keys automatically by applying the index’s key prefix and separator configuration.

NOTE: Cluster users will need to incorporate hash tags into their document IDs and only call this method with documents from a single hash tag at a time.

Parameters**ids** (*Union[str, List[str]]*) – The document ID or IDs to remove from the index.**Returns**

Count of documents deleted from Redis.

Return type

int

async drop_keys(keys)

Remove a specific entry or entries from the index by it’s key ID.

Parameters**keys** (*Union[str, List[str]]*) – The document ID or IDs to remove from the index.**Returns**

Count of records deleted from Redis.

Return type

int

async exists()

Check if the index exists in Redis.

Returns

True if the index exists, False otherwise.

Return type

bool

async expire_keys(keys, ttl)

Set the expiration time for a specific entry or entries in Redis.

Parameters

- **keys** (*Union[str, List[str]]*) – The entry ID or IDs to set the expiration for.
- **ttl** (*int*) – The time-to-live in seconds.

Return typeint | *List*[int]**async fetch(id)**

Asynchronously 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).

Parameters

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

Returns

The fetched object.

Return type

Dict[str, Any]

classmethod from_dict(schema_dict, **kwargs)

Create a SearchIndex from a dictionary.

Parameters

schema_dict (*Dict[str, Any]*) – A dictionary containing the schema.

Returns

A RedisVL SearchIndex object.

Return type*SearchIndex*

```
from redisvl.index import SearchIndex

index = SearchIndex.from_dict({
    "index": {
        "name": "my-index",
        "prefix": "rvl",
        "storage_type": "hash",
    },
    "fields": [
        {"name": "doc-id", "type": "tag"}
    ]
}, redis_url="redis://localhost:6379")
```

async classmethod from_existing(*name*, *redis_client=None*, *redis_url=None*, ***kwargs*)

Initialize from an existing search index in Redis by index name.

Parameters

- **name** (*str*) – Name of the search index in Redis.
- **redis_client** (*Optional[Redis]*) – An instantiated redis client.
- **redis_url** (*Optional[str]*) – The URL of the Redis server to connect to.

classmethod from_yaml(*schema_path*, ***kwargs*)

Create a SearchIndex from a YAML schema file.

Parameters

schema_path (*str*) – Path to the YAML schema file.

Returns

A RedisVL SearchIndex object.

Return type

SearchIndex

```
from redisvl.index import SearchIndex

index = SearchIndex.from_yaml("schemas/schema.yaml", redis_url="redis://
↪localhost:6379")
```

async info(*name=None*)

Get information about the index.

Parameters

name (*str*, *optional*) – Index name to fetch info about. Defaults to None.

Returns

A dictionary containing the information about the index.

Return type

dict

key(*id*)

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).

Parameters

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

Returns

The full Redis key including key prefix and value as a string.

Return type

str

async listall()

List all search indices in Redis database.

Returns

The list of indices in the database.

Return type

List[str]

load(*data*, *id_field=None*, *keys=None*, *ttl=None*, *preprocess=None*, *concurrency=None*, *batch_size=None*)

Asynchronously load objects to Redis. 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.

Parameters

- **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.
- **concurrency** (*int | None*)

Returns

List of keys loaded to Redis.

Return type

List[str]

Raises

- **SchemaValidationError** – If validation fails when `validate_on_load` is enabled.
- **RedisVLError** – If there's an error loading data to Redis.

```
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
def add_field(d):
    d["new_field"] = 123
    return d
keys = await index.load(data, preprocess=add_field)
```

async paginate(*query*, *page_size=30*)

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.

Parameters

- **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.

Return type

AsyncGenerator

```
# 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.

async query(query)

Asynchronously execute a query on the index.

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

Parameters

query (*Union[BaseQuery, AggregationQuery]*) – The query to run.

Returns

A list of search results.

Return type

`List[Result]`

```
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)
```

async search(*args, **kwargs)

Perform an async 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

Raw Redis search results.

Return type

Result

set_client(*redis_client*)

[DEPRECATED] Manually set the Redis client to use with the search index. This method is deprecated; please provide connection parameters in `__init__`.

Parameters

redis_client (*Redis* | *RedisCluster* | *Redis* | *RedisCluster*)

property client: *Redis* | *RedisCluster* | *None*

The underlying redis-py client object.

property key_separator: *str*

The optional separator between a defined prefix and key value in forming a Redis key.

property name: *str*

The name of the Redis search index.

property prefix: *str*

The optional key prefix that comes before a unique key value in forming a Redis key. If multiple prefixes are configured, returns the first one.

property storage_type: *StorageType*

The underlying storage type for the search index; either hash or json.

2.2.3 Vector

The Vector class in RedisVL is a container that encapsulates a numerical vector, it's datatype, corresponding index field name, and optional importance weight. It is used when constructing multi-vector queries using the MultiVectorQuery class.

Vector

class Vector(*, *vector*, *field_name*, *dtype*='float32', *weight*=1.0)

Simple object containing the necessary arguments to perform a multi vector query.

Create a new model by parsing and validating input data from keyword arguments.

Raises [*ValidationError*][*pydantic_core.ValidationError*] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **vector** (*List[float]* | *bytes*)
- **field_name** (*str*)
- **dtype** (*str*)
- **weight** (*float*)

validate_vector()

If the vector passed in is an array of float convert it to a byte string.

Return type

Self

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

2.2.4 Query

Query classes in RedisVL provide a structured way to define simple or complex queries for different use cases. Each query class wraps the `redis-py` Query module <https://github.com/redis/redis-py/blob/master/redis/commands/search/query.py> with extended functionality for ease-of-use.

VectorQuery

```
class VectorQuery(vector, vector_field_name, return_fields=None, filter_expression=None, dtype='float32',
                  num_results=10, return_score=True, dialect=2, sort_by=None, in_order=False,
                  hybrid_policy=None, batch_size=None, ef_runtime=None,
                  normalize_vector_distance=False)
```

Bases: `BaseVectorQuery`, `BaseQuery`

A query for running a vector search along with an optional filter expression.

Parameters

- **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 RedisSearch query dialect. Defaults to 2.
- **sort_by** (`Optional[SortSpec]`) – The field(s) to order the results by. Can be: - `str`: single field name - `Tuple[str, str]`: (field_name, “ASC”|“DESC”) - `List`: list of fields or tuples Note: Only the first field is used for Redis sorting. 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`.
- **hybrid_policy** (`Optional[str]`) – Controls how filters are applied during vector search. Options are “BATCHES” (paginates through small batches of nearest neighbors) or “AD-HOC_BF” (computes scores for all vectors passing the filter). “BATCHES” mode is typically

faster for queries with selective filters. “ADHOC_BF” mode is better when filters match a large portion of the dataset. Defaults to None, which lets Redis auto-select the optimal policy.

- **batch_size** (*Optional[int]*) – When `hybrid_policy` is “BATCHES”, controls the number of vectors to fetch in each batch. Larger values may improve performance at the cost of memory usage. Only applies when `hybrid_policy=“BATCHES”`. Defaults to None, which lets Redis auto-select an appropriate batch size.
- **ef_runtime** (*Optional[int]*) – Controls the size of the dynamic candidate list for HNSW algorithm at query time. Higher values improve recall at the expense of slower search performance. Defaults to None, which uses the index-defined value.
- **normalize_vector_distance** (*bool*) – Redis supports 3 distance metrics: L2 (euclidean), IP (inner product), and COSINE. By default, L2 distance returns an unbounded value. COSINE distance returns a value between 0 and 2. IP returns a value determined by the magnitude of the vector. Setting this flag to true converts COSINE and L2 distance to a similarity score between 0 and 1. Note: setting this flag to true for IP will throw a warning since by definition COSINE similarity is normalized IP.

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>

dialect (*dialect*)

Add a dialect field to the query.

- **dialect** - dialect version to execute the query under

Parameters

dialect (*int*)

Return type

Query

expander (*expander*)

Add a expander field to the query.

- **expander** - the name of the expander

Parameters

expander (*str*)

Return type

Query

in_order ()

Match only documents where the query terms appear in the same order in the document. i.e. for the query “hello world”, we do not match “world hello”

Return type

Query

language(*language*)

Analyze the query as being in the specified language.

Parameters

language (*str*) – The language (e.g. *chinese* or *english*)

Return type

Query

limit_fields(**fields*)

Limit the search to specific TEXT fields only.

- **fields**: A list of strings, case sensitive field names

from the defined schema.

Parameters

fields (*List[str]*)

Return type

Query

limit_ids(**ids*)

Limit the results to a specific set of pre-known document ids of any length.

Return type

Query

no_content()

Set the query to only return ids and not the document content.

Return type

Query

no_stopwords()

Prevent the query from being filtered for stopwords. Only useful in very big queries that you are certain contain no stopwords.

Return type

Query

paging(*offset, num*)

Set the paging for the query (defaults to 0..10).

- **offset**: Paging offset for the results. Defaults to 0
- **num**: How many results do we want

Parameters

- **offset** (*int*)

- **num** (*int*)

Return type

Query

query_string()

Return the query string of this query only.

Return type

str

return_fields(*fields, skip_decode=None)

Set the fields to return with search results.

Parameters

- ***fields** – Variable number of field names to return.
- **skip_decode** (*str* | *List[str]* | *None*) – Optional field name or list of field names that should not be decoded. Useful for binary data like embeddings.

Returns

Returns the query object for method chaining.

Return type

self

Raises

TypeError – If skip_decode is not a string, list, or None.

scorer(scorer)

Use a different scoring function to evaluate document relevance. Default is *TFIDF*.

Since Redis 8.0 default was changed to *BM25STD*.

Parameters

scorer (*str*) – The scoring function to use (e.g. *TFIDF.DOCNORM* or *BM25*)

Return type

Query

set_batch_size(batch_size)

Set the batch size for the query.

Parameters

batch_size (*int*) – The batch size to use when hybrid_policy is “BATCHES”.

Raises

- **TypeError** – If batch_size is not an integer
- **ValueError** – If batch_size is not positive

set_ef_runtime(ef_runtime)

Set the EF_RUNTIME parameter for the query.

Parameters

ef_runtime (*int*) – The EF_RUNTIME value to use for HNSW algorithm. Higher values improve recall at the expense of slower search.

Raises

- **TypeError** – If ef_runtime is not an integer
- **ValueError** – If ef_runtime is not positive

set_filter(filter_expression=None)

Set the filter expression for the query.

Parameters

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.

set_hybrid_policy(*hybrid_policy*)

Set the hybrid policy for the query.

Parameters

hybrid_policy (*str*) – The hybrid policy to use. Options are “BATCHES” or “AD-HOC_BF”.

Raises

ValueError – If *hybrid_policy* is not one of the valid options

slop(*slop*)

Allow a maximum of N intervening non matched terms between phrase terms (0 means exact phrase).

Parameters

slop (*int*)

Return type

Query

sort_by(*sort_spec=None, asc=True*)

Set the sort order for query results.

This method supports sorting by single or multiple fields. Note that Redis Search natively supports only a single SORTBY field. When multiple fields are specified, only the FIRST field is used for the Redis SORTBY clause.

Parameters

- **sort_spec** (*str | Tuple[str, str] | List[str | Tuple[str, str]] | None*) – Sort specification in various formats: - *str*: single field name - *Tuple[str, str]*: (*field_name*, “ASC”|”DESC”) - *List*: list of field names or tuples
- **asc** (*bool*) – Default sort direction when not specified (only used when *sort_spec* is a string). Defaults to True (ascending).

Returns

Returns the query object for method chaining.

Return type

self

Raises

- **TypeError** – If *sort_spec* is not a valid type.
- **ValueError** – If direction is not “ASC” or “DESC”.

Examples

```
>>> query.sort_by("price") # Single field, ascending
>>> query.sort_by(("price", "DESC")) # Single field, descending
>>> query.sort_by(["price", "rating"]) # Multiple fields (only first used)
>>> query.sort_by([("price", "DESC"), ("rating", "ASC")])
```

Note

When multiple fields are specified, only the first field is used for sorting in Redis. Future versions may support multi-field sorting through post-query sorting in Python.

timeout (*timeout*)

overrides the timeout parameter of the module

Parameters

timeout (*float*)

Return type

Query

verbatim()

Set the query to be verbatim, i.e. use no query expansion or stemming.

Return type

Query

with_payloads()

Ask the engine to return document payloads.

Return type

Query

with_scores()

Ask the engine to return document search scores.

Return type

Query

property batch_size: int | None

Return the batch size for the query.

Returns

The batch size for the query.

Return type

Optional[int]

property ef_runtime: int | None

Return the EF_RUNTIME parameter for the query.

Returns

The EF_RUNTIME value for the query.

Return type

Optional[int]

property filter: str | *FilterExpression*

The filter expression for the query.

property hybrid_policy: str | None

Return the hybrid policy for the query.

Returns

The hybrid policy for the query.

Return type

Optional[str]

property params: Dict[str, Any]

Return the parameters for the query.

Returns

The parameters for the query.

Return type

Dict[str, Any]

property query: BaseQuery

Return self as the query object.

VectorRangeQuery

```
class VectorRangeQuery(vector, vector_field_name, return_fields=None, filter_expression=None,
                       dtype='float32', distance_threshold=0.2, epsilon=None, num_results=10,
                       return_score=True, dialect=2, sort_by=None, in_order=False, hybrid_policy=None,
                       batch_size=None, normalize_vector_distance=False)
```

Bases: BaseVectorQuery, BaseQuery

A query for running a filtered vector search based on semantic distance threshold.

Parameters

- **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** (*float*) – The threshold for vector distance. A smaller threshold indicates a stricter semantic search. Defaults to 0.2.
- **epsilon** (*Optional[float]*) – The relative factor for vector range queries, setting boundaries for candidates within radius * (1 + epsilon). This controls how extensive the search is beyond the specified radius. Higher values increase recall at the expense of performance. Defaults to None, which uses the index-defined epsilon (typically 0.01).
- **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[SortSpec]*) – The field(s) to order the results by. Can be: - *str*: single field name - *Tuple[str, str]*: (field_name, “ASC”|“DESC”) - *List*: list of fields or tuples Note: Only the first field is used for Redis sorting. 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.
- **hybrid_policy** (*Optional[str]*) – Controls how filters are applied during vector search. Options are “BATCHES” (paginates through small batches of nearest neighbors) or “AD-HOC_BF” (computes scores for all vectors passing the filter). “BATCHES” mode is typically faster for queries with selective filters. “ADHOC_BF” mode is better when filters match a large portion of the dataset. Defaults to None, which lets Redis auto-select the optimal policy.
- **batch_size** (*Optional[int]*) – When hybrid_policy is “BATCHES”, controls the number of vectors to fetch in each batch. Larger values may improve performance at the cost of

memory usage. Only applies when `hybrid_policy="BATCHES"`. Defaults to `None`, which lets Redis auto-select an appropriate batch size.

- **normalize_vector_distance** (*bool*) – Redis supports 3 distance metrics: L2 (euclidean), IP (inner product), and COSINE. By default, L2 distance returns an unbounded value. COSINE distance returns a value between 0 and 2. IP returns a value determined by the magnitude of the vector. Setting this flag to true converts COSINE and L2 distance to a similarity score between 0 and 1. Note: setting this flag to true for IP will throw a warning since by definition COSINE similarity is normalized IP.

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>

dialect(*dialect*)

Add a dialect field to the query.

- **dialect** - dialect version to execute the query under

Parameters

dialect (*int*)

Return type

Query

expander(*expander*)

Add a expander field to the query.

- **expander** - the name of the expander

Parameters

expander (*str*)

Return type

Query

in_order()

Match only documents where the query terms appear in the same order in the document. i.e. for the query “hello world”, we do not match “world hello”

Return type

Query

language(*language*)

Analyze the query as being in the specified language.

Parameters

language (*str*) – The language (e.g. *chinese* or *english*)

Return type

Query

limit_fields(*fields)

Limit the search to specific TEXT fields only.

- **fields**: A list of strings, case sensitive field names

from the defined schema.

Parameters

fields (*List[str]*)

Return type

Query

limit_ids(*ids)

Limit the results to a specific set of pre-known document ids of any length.

Return type

Query

no_content()

Set the query to only return ids and not the document content.

Return type

Query

no_stopwords()

Prevent the query from being filtered for stopwords. Only useful in very big queries that you are certain contain no stopwords.

Return type

Query

paging(offset, num)

Set the paging for the query (defaults to 0..10).

- **offset**: Paging offset for the results. Defaults to 0
- **num**: How many results do we want

Parameters

- **offset** (*int*)

- **num** (*int*)

Return type

Query

query_string()

Return the query string of this query only.

Return type

str

return_fields(*fields, skip_decode=None)

Set the fields to return with search results.

Parameters

- ***fields** – Variable number of field names to return.
- **skip_decode** (*str | List[str] | None*) – Optional field name or list of field names that should not be decoded. Useful for binary data like embeddings.

Returns

Returns the query object for method chaining.

Return type

`self`

Raises

TypeError – If `skip_decode` is not a string, list, or `None`.

scorer(*scorer*)

Use a different scoring function to evaluate document relevance. Default is *TFIDF*.

Since Redis 8.0 default was changed to *BM25STD*.

Parameters

scorer (*str*) – The scoring function to use (e.g. *TFIDF.DOCNORM* or *BM25*)

Return type

Query

set_batch_size(*batch_size*)

Set the batch size for the query.

Parameters

batch_size (*int*) – The batch size to use when `hybrid_policy` is “BATCHES”.

Raises

- **TypeError** – If `batch_size` is not an integer
- **ValueError** – If `batch_size` is not positive

set_distance_threshold(*distance_threshold*)

Set the distance threshold for the query.

Parameters

distance_threshold (*float*) – Vector distance threshold.

Raises

- **TypeError** – If `distance_threshold` is not a float or int
- **ValueError** – If `distance_threshold` is negative

set_epsilon(*epsilon*)

Set the epsilon parameter for the range query.

Parameters

epsilon (*float*) – The relative factor for vector range queries, setting boundaries for candidates within `radius * (1 + epsilon)`.

Raises

- **TypeError** – If `epsilon` is not a float or int
- **ValueError** – If `epsilon` is negative

set_filter(*filter_expression=None*)

Set the filter expression for the query.

Parameters

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.

set_hybrid_policy(*hybrid_policy*)

Set the hybrid policy for the query.

Parameters

hybrid_policy (*str*) – The hybrid policy to use. Options are “BATCHES” or “AD-HOC_BF”.

Raises

ValueError – If `hybrid_policy` is not one of the valid options

slop(*slop*)

Allow a maximum of *N* intervening non matched terms between phrase terms (0 means exact phrase).

Parameters

slop (*int*)

Return type

Query

sort_by(*sort_spec=None, asc=True*)

Set the sort order for query results.

This method supports sorting by single or multiple fields. Note that Redis Search natively supports only a single SORTBY field. When multiple fields are specified, only the FIRST field is used for the Redis SORTBY clause.

Parameters

- **sort_spec** (*str | Tuple[str, str] | List[str | Tuple[str, str]] | None*) – Sort specification in various formats: - *str*: single field name - *Tuple[str, str]*: (*field_name*, “ASC”|“DESC”) - *List*: list of field names or tuples
- **asc** (*bool*) – Default sort direction when not specified (only used when `sort_spec` is a string). Defaults to `True` (ascending).

Returns

Returns the query object for method chaining.

Return type

self

Raises

- **TypeError** – If `sort_spec` is not a valid type.
- **ValueError** – If direction is not “ASC” or “DESC”.

Examples

```
>>> query.sort_by("price") # Single field, ascending
>>> query.sort_by(("price", "DESC")) # Single field, descending
>>> query.sort_by(["price", "rating"]) # Multiple fields (only first used)
>>> query.sort_by([("price", "DESC"), ("rating", "ASC")])
```

Note

When multiple fields are specified, only the first field is used for sorting in Redis. Future versions may support multi-field sorting through post-query sorting in Python.

timeout (*timeout*)

overrides the timeout parameter of the module

Parameters

timeout (*float*)

Return type

Query

verbatim()

Set the query to be verbatim, i.e. use no query expansion or stemming.

Return type

Query

with_payloads()

Ask the engine to return document payloads.

Return type

Query

with_scores()

Ask the engine to return document search scores.

Return type

Query

property batch_size: int | None

Return the batch size for the query.

Returns

The batch size for the query.

Return type

Optional[int]

property distance_threshold: float

Return the distance threshold for the query.

Returns

The distance threshold for the query.

Return type

float

property epsilon: float | None

Return the epsilon for the query.

Returns

The epsilon for the query, or None if not set.

Return type

Optional[float]

property filter: `str` | *FilterExpression*

The filter expression for the query.

property hybrid_policy: `str` | `None`

Return the hybrid policy for the query.

Returns

The hybrid policy for the query.

Return type

`Optional[str]`

property params: `Dict[str, Any]`

Return the parameters for the query.

Returns

The parameters for the query.

Return type

`Dict[str, Any]`

property query: `BaseQuery`

Return self as the query object.

HybridQuery

class HybridQuery(*args, **kwargs)

Bases: `AggregateHybridQuery`

Backward compatibility wrapper for `AggregateHybridQuery`.

Deprecated since version HybridQuery: is a backward compatibility wrapper around `AggregateHybridQuery` and will eventually be replaced with a new hybrid query implementation. To maintain current functionality please use `AggregateHybridQuery` directly.”

Instantiates a `AggregateHybridQuery` object.

Parameters

- **text** (*str*) – The text to search for.
- **text_field_name** (*str*) – The text field name to search in.
- **vector** (*Union[bytes, List[float]]*) – The vector to perform vector similarity search.
- **vector_field_name** (*str*) – The vector field name to search in.
- **text_scorer** (*str, optional*) – The text scorer to use. Options are {TFIDF, TFIDF.DOCNORM, BM25, DISMAX, DOCSCORE, BM25STD}. Defaults to “BM25STD”.
- **filter_expression** (*Optional[FilterExpression], optional*) – The filter expression to use. Defaults to None.
- **alpha** (*float, optional*) – The weight of the vector similarity. Documents will be scored as: $\text{hybrid_score} = (\text{alpha}) * \text{vector_score} + (1 - \text{alpha}) * \text{text_score}$. Defaults to 0.7.
- **dtype** (*str, optional*) – The data type of the vector. Defaults to “float32”.
- **num_results** (*int, optional*) – The number of results to return. Defaults to 10.
- **return_fields** (*Optional[List[str]], optional*) – The fields to return. Defaults to None.

- **stopwords** (*Optional[Union[str, Set[str]]], optional*) – The stopwords to remove from the provided text prior to search. If a string such as “english” “german” is provided then a default set of stopwords for that language will be used. if a list, set, or tuple of strings is provided then those will be used as stopwords. Defaults to “english”. if set to “None” then no stopwords will be removed.
- **dialect** (*int, optional*) – The Redis dialect version. Defaults to 2.
- **text_weights** (*Optional[Dict[str, float]]*) – The importance weighting of individual words within the query text. Defaults to None, as no modifications will be made to the text_scorer score.

Raises

- **ValueError** – If the text string is empty, or if the text string becomes empty after stopwords are removed.
- **TypeError** – If the stopwords are not a set, list, or tuple of strings.

add_scores()

If set, includes the score as an ordinary field of the row.

Return type

AggregateRequest

apply(kwexpr)**

Specify one or more projection expressions to add to each result

Parameters

- **kwexpr: One or more key-value pairs for a projection. The key is** the alias for the projection, and the value is the projection expression itself, for example *apply(square_root=sqrt(@foo))*

Return type

AggregateRequest

dialect(dialect)

Add a dialect field to the aggregate command.

- **dialect** - dialect version to execute the query under

Parameters

dialect (*int*)

Return type

AggregateRequest

filter(expressions)

Specify filter for post-query results using predicates relating to values in the result set.

Parameters

- **fields: Fields to group by. This can either be a single string,** or a list of strings.

Parameters

expressions (*str | List[str]*)

Return type

AggregateRequest

group_by(*fields*, **reducers*)

Specify by which fields to group the aggregation.

Parameters

- **fields:** Fields to group by. This can either be a single string, or a list of strings. both cases, the field should be specified as *@field*.
- **reducers:** One or more reducers. Reducers may be found in the *aggregation* module.

Parameters

- **fields** (*List[str]*)
- **reducers** (*Reducer | List[Reducer]*)

Return type

AggregateRequest

limit(*offset*, *num*)

Sets the limit for the most recent group or query.

If no group has been defined yet (via *group_by()*) then this sets the limit for the initial pool of results from the query. Otherwise, this limits the number of items operated on from the previous group.

Setting a limit on the initial search results may be useful when attempting to execute an aggregation on a sample of a large data set.

Parameters

- **offset:** Result offset from which to begin paging
- **num:** Number of results to return

Example of sorting the initial results:

```
` AggregateRequest("@sale_amount:[10000, inf]") .limit(0, 10) .group_by("@state", r.count()) `
```

Will only group by the states found in the first 10 results of the query *@sale_amount:[10000, inf]*. On the other hand,

```
` AggregateRequest("@sale_amount:[10000, inf]") .limit(0, 1000) .group_by("@state", r.count()) .limit(0, 10) `
```

Will group all the results matching the query, but only return the first 10 groups.

If you only wish to return a *top-N* style query, consider using *sort_by()* instead.

Parameters

- **offset** (*int*)
- **num** (*int*)

Return type

AggregateRequest

load(**fields*)

Indicate the fields to be returned in the response. These fields are returned in addition to any others implicitly specified.

Parameters

- **fields**: If fields not specified, all the fields will be loaded.

Otherwise, fields should be given in the format of *@field*.

Parameters

fields (*str*)

Return type

AggregateRequest

scorer(*scorer*)

Use a different scoring function to evaluate document relevance. Default is *TFIDF*.

Parameters

scorer (*str*) – The scoring function to use (e.g. *TFIDF.DOCNORM* or *BM25*)

Return type

AggregateRequest

set_text_weights(*weights*)

Set or update the text weights for the query.

Parameters

- **text_weights** – Dictionary of word:weight mappings
- **weights** (*Dict[str, float]*)

sort_by(**fields, **kwargs*)

Indicate how the results should be sorted. This can also be used for *top-N* style queries

Parameters

- **fields**: **The fields by which to sort. This can be either a single** field or a list of fields. If you wish to specify order, you can use the *Asc* or *Desc* wrapper classes.
- **max**: **Maximum number of results to return. This can be** used instead of *LIMIT* and is also faster.

Example of sorting by *foo* ascending and *bar* descending:

```
` sort_by(Asc("@foo"), Desc("@bar")) `
```

Return the top 10 customers:

```
` AggregateRequest().group_by("@customer", r.sum("@paid").alias(FIELDNAME))\n.sort_by(Desc("@paid"), max=10) `
```

Parameters

fields (*str*)

Return type

AggregateRequest

with_schema()

If set, the *schema* property will contain a list of [*field*, *type*] entries in the result object.

Return type

AggregateRequest

property params: *Dict[str, Any]*

Return the parameters for the aggregation.

Returns

The parameters for the aggregation.

Return type

Dict[str, Any]

property stopwords: Set[str]

Return the stopwords used in the query. :returns: The stopwords used in the query. :rtype: Set[str]

property text_weights: Dict[str, float]

Get the text weights.

Returns

weight mappings.

Return type

Dictionary of word

TextQuery

```
class TextQuery(text, text_field_name, text_scorer='BM25STD', filter_expression=None, return_fields=None,
                num_results=10, return_score=True, dialect=2, sort_by=None, in_order=False, params=None,
                stopwords='english', text_weights=None)
```

Bases: BaseQuery

TextQuery is a query for running a full text search, along with an optional filter expression.

```
from redisvl.query import TextQuery
from redisvl.index import SearchIndex

index = SearchIndex.from_yaml(index.yaml)

query = TextQuery(
    text="example text",
    text_field_name="text_field",
    text_scorer="BM25STD",
    filter_expression=None,
    num_results=10,
    return_fields=["field1", "field2"],
    stopwords="english",
    dialect=2,
)

results = index.query(query)
```

A query for running a full text search, along with an optional filter expression.

Parameters

- **text** (*str*) – The text string to perform the text search with.
- **text_field_name** (*Union[str, Dict[str, float]]*) – The name of the document field to perform text search on, or a dictionary mapping field names to their weights.
- **text_scorer** (*str, optional*) – The text scoring algorithm to use. Defaults to BM25STD. Options are {TFIDF, BM25STD, BM25, TFIDF.DOCNORM, DIS-MAX, DOCSCORE}. See <https://redis.io/docs/latest/develop/interact/search-and-query/advanced-concepts/scoring/>
- **filter_expression** (*Union[str, FilterExpression], optional*) – A filter to apply along with the text search. Defaults to None.

- **return_fields** (*List[str]*) – The declared fields to return with search results.
- **num_results** (*int, optional*) – The top k results to return from the search. Defaults to 10.
- **return_score** (*bool, optional*) – Whether to return the text score. Defaults to True.
- **dialect** (*int, optional*) – The RediSearch query dialect. Defaults to 2.
- **sort_by** (*Optional[SortSpec]*) – The field(s) to order the results by. Can be: - str: single field name - Tuple[str, str]: (field_name, “ASC”|”DESC”) - List: list of fields or tuples Note: Only the first field is used for Redis sorting. Defaults to None. Results will be ordered by text score.
- **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.
- **params** (*Optional[Dict[str, Any]], optional*) – The parameters for the query. Defaults to None.
- **stopwords** (*Optional[Union[str, Set[str]]]*) – The set of stop words to remove from the query text. If a language like ‘english’ or ‘spanish’ is provided a default set of stopwords for that language will be used. Users may specify their own stop words by providing a List or Set of words. if set to None, then no words will be removed. Defaults to ‘english’.
- **text_weights** (*Optional[Dict[str, float]]*) – The importance weighting of individual words within the query text. Defaults to None, as no modifications will be made to the text_scorer score.

Raises

- **ValueError** – if stopwords language string cannot be loaded.
- **TypeError** – If stopwords is not a valid iterable set of strings.

dialect (*dialect*)

Add a dialect field to the query.

- **dialect** - dialect version to execute the query under

Parameters

dialect (*int*)

Return type

Query

expander (*expander*)

Add a expander field to the query.

- **expander** - the name of the expander

Parameters

expander (*str*)

Return type

Query

in_order ()

Match only documents where the query terms appear in the same order in the document. i.e. for the query “hello world”, we do not match “world hello”

Return type*Query***language**(*language*)

Analyze the query as being in the specified language.

Parameters

language (*str*) – The language (e.g. *chinese* or *english*)

Return type*Query***limit_fields**(**fields*)

Limit the search to specific TEXT fields only.

- **fields**: A list of strings, case sensitive field names

from the defined schema.

Parameters

fields (*List[str]*)

Return type*Query***limit_ids**(**ids*)

Limit the results to a specific set of pre-known document ids of any length.

Return type*Query***no_content**()

Set the query to only return ids and not the document content.

Return type*Query***no_stopwords**()

Prevent the query from being filtered for stopwords. Only useful in very big queries that you are certain contain no stopwords.

Return type*Query***paging**(*offset, num*)

Set the paging for the query (defaults to 0..10).

- **offset**: Paging offset for the results. Defaults to 0
- **num**: How many results do we want

Parameters

- **offset** (*int*)

- **num** (*int*)

Return type*Query*

query_string()

Return the query string of this query only.

Return type

str

return_fields(*fields, skip_decode=None)

Set the fields to return with search results.

Parameters

- ***fields** – Variable number of field names to return.
- **skip_decode** (*str* | *List[str]* | *None*) – Optional field name or list of field names that should not be decoded. Useful for binary data like embeddings.

Returns

Returns the query object for method chaining.

Return type

self

Raises

TypeError – If skip_decode is not a string, list, or None.

scorer(scorer)

Use a different scoring function to evaluate document relevance. Default is *TFIDF*.

Since Redis 8.0 default was changed to *BM25STD*.

Parameters

scorer (*str*) – The scoring function to use (e.g. *TFIDF.DOCNORM* or *BM25*)

Return type

Query

set_field_weights(field_weights)

Set or update the field weights for the query.

Parameters

field_weights (*str* | *Dict[str, float]*) – Either a single field name or dictionary of field:weight mappings

set_filter(filter_expression=None)

Set the filter expression for the query.

Parameters

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.

set_text_weights(weights)

Set or update the text weights for the query.

Parameters

- **text_weights** – Dictionary of word:weight mappings
- **weights** (*Dict[str, float]*)

slop(*slop*)

Allow a maximum of N intervening non matched terms between phrase terms (0 means exact phrase).

Parameters

slop (*int*)

Return type

Query

sort_by(*sort_spec=None, asc=True*)

Set the sort order for query results.

This method supports sorting by single or multiple fields. Note that Redis Search natively supports only a single SORTBY field. When multiple fields are specified, only the FIRST field is used for the Redis SORTBY clause.

Parameters

- **sort_spec** (*str | Tuple[str, str] | List[str | Tuple[str, str]] | None*) – Sort specification in various formats: - *str*: single field name - *Tuple[str, str]*: (field_name, “ASC”|”DESC”) - *List*: list of field names or tuples
- **asc** (*bool*) – Default sort direction when not specified (only used when *sort_spec* is a string). Defaults to True (ascending).

Returns

Returns the query object for method chaining.

Return type

self

Raises

- **TypeError** – If *sort_spec* is not a valid type.
- **ValueError** – If direction is not “ASC” or “DESC”.

Examples

```
>>> query.sort_by("price") # Single field, ascending
>>> query.sort_by(("price", "DESC")) # Single field, descending
>>> query.sort_by(["price", "rating"]) # Multiple fields (only first used)
>>> query.sort_by([("price", "DESC"), ("rating", "ASC")])
```

Note

When multiple fields are specified, only the first field is used for sorting in Redis. Future versions may support multi-field sorting through post-query sorting in Python.

timeout(*timeout*)

overrides the timeout parameter of the module

Parameters

timeout (*float*)

Return type

Query

verbatim()

Set the query to be verbatim, i.e. use no query expansion or stemming.

Return type

Query

with_payloads()

Ask the engine to return document payloads.

Return type

Query

with_scores()

Ask the engine to return document search scores.

Return type

Query

property field_weights: Dict[str, float]

Get the field weights for the query.

Returns

Dictionary mapping field names to their weights

property filter: str | *FilterExpression*

The filter expression for the query.

property params: Dict[str, Any]

Return the query parameters.

property query: BaseQuery

Return self as the query object.

property text_field_name: str | Dict[str, float]

Get the text field name(s) - for backward compatibility.

Returns

Either a single field name string (if only one field with weight 1.0) or a dictionary of field:weight mappings.

property text_weights: Dict[str, float]

Get the text weights.

Returns

weight mappings.

Return type

Dictionary of word

FilterQuery

```
class FilterQuery(filter_expression=None, return_fields=None, num_results=10, dialect=2, sort_by=None,
                  in_order=False, params=None)
```

Bases: BaseQuery

A query for running a filtered search with a filter expression.

Parameters

- **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[SortSpec], optional*) – The field(s) to order the results by. Can be: - str: single field name (e.g., “price”) - Tuple[str, str]: (field_name, “ASC”|”DESC”) (e.g., (“price”, “DESC”)) - List: list of fields or tuples (e.g., [“price”, (“rating”, “DESC”)]) Note: Redis Search only supports single-field sorting, so only the first field is used. 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

dialect(*dialect*)

Add a dialect field to the query.

- **dialect** - dialect version to execute the query under

Parameters

dialect (*int*)

Return type

Query

expander(*expander*)

Add a expander field to the query.

- **expander** - the name of the expander

Parameters

expander (*str*)

Return type

Query

in_order()

Match only documents where the query terms appear in the same order in the document. i.e. for the query “hello world”, we do not match “world hello”

Return type

Query

language(*language*)

Analyze the query as being in the specified language.

Parameters

language (*str*) – The language (e.g. *chinese* or *english*)

Return type*Query***limit_fields(*fields)**

Limit the search to specific TEXT fields only.

- **fields**: A list of strings, case sensitive field names

from the defined schema.

Parameters**fields** (*List[str]*)**Return type***Query***limit_ids(*ids)**

Limit the results to a specific set of pre-known document ids of any length.

Return type*Query***no_content()**

Set the query to only return ids and not the document content.

Return type*Query***no_stopwords()**

Prevent the query from being filtered for stopwords. Only useful in very big queries that you are certain contain no stopwords.

Return type*Query***paging(offset, num)**

Set the paging for the query (defaults to 0..10).

- **offset**: Paging offset for the results. Defaults to 0
- **num**: How many results do we want

Parameters

- **offset** (*int*)
- **num** (*int*)

Return type*Query***query_string()**

Return the query string of this query only.

Return type*str***return_fields(*fields, skip_decode=None)**

Set the fields to return with search results.

Parameters

- ***fields** – Variable number of field names to return.

- **skip_decode** (*str* | *List[str]* | *None*) – Optional field name or list of field names that should not be decoded. Useful for binary data like embeddings.

Returns

Returns the query object for method chaining.

Return type

self

Raises

TypeError – If skip_decode is not a string, list, or None.

scorer(*scorer*)

Use a different scoring function to evaluate document relevance. Default is *TFIDF*.

Since Redis 8.0 default was changed to BM25STD.

Parameters

scorer (*str*) – The scoring function to use (e.g. *TFIDF.DOCNORM* or *BM25*)

Return type

Query

set_filter(*filter_expression=None*)

Set the filter expression for the query.

Parameters

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.

slop(*slop*)

Allow a maximum of N intervening non matched terms between phrase terms (0 means exact phrase).

Parameters

slop (*int*)

Return type

Query

sort_by(*sort_spec=None, asc=True*)

Set the sort order for query results.

This method supports sorting by single or multiple fields. Note that Redis Search natively supports only a single SORTBY field. When multiple fields are specified, only the FIRST field is used for the Redis SORTBY clause.

Parameters

- **sort_spec** (*str* | *Tuple[str, str]* | *List[str | Tuple[str, str]]* | *None*) – Sort specification in various formats: - *str*: single field name - *Tuple[str, str]*: (field_name, “ASC”|”DESC”) - *List*: list of field names or tuples
- **asc** (*bool*) – Default sort direction when not specified (only used when sort_spec is a string). Defaults to True (ascending).

Returns

Returns the query object for method chaining.

Return type

self

Raises

- **TypeError** – If `sort_spec` is not a valid type.
- **ValueError** – If `direction` is not “ASC” or “DESC”.

Examples

```
>>> query.sort_by("price") # Single field, ascending
>>> query.sort_by(("price", "DESC")) # Single field, descending
>>> query.sort_by(["price", "rating"]) # Multiple fields (only first used)
>>> query.sort_by([("price", "DESC"), ("rating", "ASC")])
```

Note

When multiple fields are specified, only the first field is used for sorting in Redis. Future versions may support multi-field sorting through post-query sorting in Python.

timeout (*timeout*)

overrides the timeout parameter of the module

Parameters

timeout (*float*)

Return type

Query

verbatim()

Set the query to be verbatim, i.e. use no query expansion or stemming.

Return type

Query

with_payloads()

Ask the engine to return document payloads.

Return type

Query

with_scores()

Ask the engine to return document search scores.

Return type

Query

property filter: **str** | *FilterExpression*

The filter expression for the query.

property params: **Dict**[**str**, **Any**]

Return the query parameters.

property query: **BaseQuery**

Return self as the query object.

CountQuery

class CountQuery(*filter_expression=None, dialect=2, params=None*)

Bases: BaseQuery

A query for a simple count operation provided some filter expression.

Parameters

- **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.
- **dialect** (*int*)

Raises

TypeError – If filter_expression is not of type redisvl.query.FilterExpression

```
from redisvl.query import CountQuery
from redisvl.query.filter import Tag

t = Tag("brand") == "Nike"
query = CountQuery(filter_expression=t)

count = index.query(query)
```

dialect(*dialect*)

Add a dialect field to the query.

- **dialect** - dialect version to execute the query under

Parameters

dialect (*int*)

Return type

Query

expander(*expander*)

Add a expander field to the query.

- **expander** - the name of the expander

Parameters

expander (*str*)

Return type

Query

in_order()

Match only documents where the query terms appear in the same order in the document. i.e. for the query “hello world”, we do not match “world hello”

Return type

Query

language(*language*)

Analyze the query as being in the specified language.

Parameters

language (*str*) – The language (e.g. *chinese* or *english*)

Return type

Query

limit_fields(**fields*)

Limit the search to specific TEXT fields only.

- **fields**: A list of strings, case sensitive field names

from the defined schema.

Parameters

fields (*List[str]*)

Return type

Query

limit_ids(**ids*)

Limit the results to a specific set of pre-known document ids of any length.

Return type

Query

no_content()

Set the query to only return ids and not the document content.

Return type

Query

no_stopwords()

Prevent the query from being filtered for stopwords. Only useful in very big queries that you are certain contain no stopwords.

Return type

Query

paging(*offset, num*)

Set the paging for the query (defaults to 0..10).

- **offset**: Paging offset for the results. Defaults to 0
- **num**: How many results do we want

Parameters

- **offset** (*int*)

- **num** (*int*)

Return type

Query

query_string()

Return the query string of this query only.

Return type

str

return_fields(*fields, skip_decode=None)

Set the fields to return with search results.

Parameters

- ***fields** – Variable number of field names to return.
- **skip_decode** (*str* | *List[str]* | *None*) – Optional field name or list of field names that should not be decoded. Useful for binary data like embeddings.

Returns

Returns the query object for method chaining.

Return type

self

Raises

TypeError – If skip_decode is not a string, list, or None.

scorer(*scorer*)

Use a different scoring function to evaluate document relevance. Default is *TFIDF*.

Since Redis 8.0 default was changed to *BM25STD*.

Parameters

scorer (*str*) – The scoring function to use (e.g. *TFIDF.DOCNORM* or *BM25*)

Return type

Query

set_filter(*filter_expression=None*)

Set the filter expression for the query.

Parameters

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.

slop(*slop*)

Allow a maximum of N intervening non matched terms between phrase terms (0 means exact phrase).

Parameters

slop (*int*)

Return type

Query

sort_by(*sort_spec=None, asc=True*)

Set the sort order for query results.

This method supports sorting by single or multiple fields. Note that Redis Search natively supports only a single SORTBY field. When multiple fields are specified, only the FIRST field is used for the Redis SORTBY clause.

Parameters

- **sort_spec** (*str* | *Tuple[str, str]* | *List[str | Tuple[str, str]]* | *None*) – Sort specification in various formats: - *str*: single field name - *Tuple[str, str]*: (field_name, “ASC”|“DESC”) - *List*: list of field names or tuples

- **asc** (*bool*) – Default sort direction when not specified (only used when `sort_spec` is a string). Defaults to True (ascending).

Returns

Returns the query object for method chaining.

Return type

self

Raises

- **TypeError** – If `sort_spec` is not a valid type.
- **ValueError** – If direction is not “ASC” or “DESC”.

Examples

```
>>> query.sort_by("price") # Single field, ascending
>>> query.sort_by(("price", "DESC")) # Single field, descending
>>> query.sort_by(["price", "rating"]) # Multiple fields (only first used)
>>> query.sort_by([("price", "DESC"), ("rating", "ASC")])
```

Note

When multiple fields are specified, only the first field is used for sorting in Redis. Future versions may support multi-field sorting through post-query sorting in Python.

timeout (*timeout*)

overrides the timeout parameter of the module

Parameters

timeout (*float*)

Return type

Query

verbatim()

Set the query to be verbatim, i.e. use no query expansion or stemming.

Return type

Query

with_payloads()

Ask the engine to return document payloads.

Return type

Query

with_scores()

Ask the engine to return document search scores.

Return type

Query

property filter: str | *FilterExpression*

The filter expression for the query.

property `params: Dict[str, Any]`

Return the query parameters.

property `query: BaseQuery`

Return self as the query object.

MultiVectorQuery

class `MultiVectorQuery(vectors, return_fields=None, filter_expression=None, num_results=10, dialect=2)`

Bases: `AggregationQuery`

MultiVectorQuery allows for search over multiple vector fields in a document simultaneously. The final score will be a weighted combination of the individual vector similarity scores following the formula:

$\text{score} = (w_1 * \text{score}_1 + w_2 * \text{score}_2 + w_3 * \text{score}_3 + \dots)$

Vectors may be of different size and datatype, but must be indexed using the 'cosine' distance_metric.

```
from redisvl.query import MultiVectorQuery, Vector
from redisvl.index import SearchIndex

index = SearchIndex.from_yaml("path/to/index.yaml")

vector_1 = Vector(
    vector=[0.1, 0.2, 0.3],
    field_name="text_vector",
    dtype="float32",
    weight=0.7,
)
vector_2 = Vector(
    vector=[0.5, 0.5],
    field_name="image_vector",
    dtype="bfloat16",
    weight=0.2,
)
vector_3 = Vector(
    vector=[0.1, 0.2, 0.3],
    field_name="text_vector",
    dtype="float64",
    weight=0.5,
)

query = MultiVectorQuery(
    vectors=[vector_1, vector_2, vector_3],
    filter_expression=None,
    num_results=10,
    return_fields=["field1", "field2"],
    dialect=2,
)

results = index.query(query)
```

Instantiates a MultiVectorQuery object.

Parameters

- **vectors** (*Union[Vector, List[Vector]]*) – The Vectors to perform vector similarity search.
- **return_fields** (*Optional[List[str]], optional*) – The fields to return. Defaults to None.
- **filter_expression** (*Optional[Union[str, FilterExpression]]*) – The filter expression to use. Defaults to None.
- **num_results** (*int, optional*) – The number of results to return. Defaults to 10.
- **dialect** (*int, optional*) – The Redis dialect version. Defaults to 2.

add_scores()

If set, includes the score as an ordinary field of the row.

Return type

AggregateRequest

apply(kwexpr)**

Specify one or more projection expressions to add to each result

Parameters

- **kwexpr:** One or more key-value pairs for a projection. The key is the alias for the projection, and the value is the projection expression itself, for example *apply(square_root=sqrt(@foo))*

Return type

AggregateRequest

dialect(dialect)

Add a dialect field to the aggregate command.

- **dialect** - dialect version to execute the query under

Parameters

dialect (*int*)

Return type

AggregateRequest

filter(expressions)

Specify filter for post-query results using predicates relating to values in the result set.

Parameters

- **fields:** Fields to group by. This can either be a single string, or a list of strings.

Parameters

expressions (*str | List[str]*)

Return type

AggregateRequest

group_by(fields, *reducers)

Specify by which fields to group the aggregation.

Parameters

- **fields:** Fields to group by. This can either be a single string, or a list of strings. both cases, the field should be specified as *@field*.
- **reducers:** One or more reducers. Reducers may be found in the *aggregation* module.

Parameters

- **fields** (*List[str]*)
- **reducers** (*Reducer | List[Reducer]*)

Return type

AggregateRequest

limit(*offset, num*)

Sets the limit for the most recent group or query.

If no group has been defined yet (via *group_by()*) then this sets the limit for the initial pool of results from the query. Otherwise, this limits the number of items operated on from the previous group.

Setting a limit on the initial search results may be useful when attempting to execute an aggregation on a sample of a large data set.

Parameters

- **offset:** Result offset from which to begin paging
- **num:** Number of results to return

Example of sorting the initial results:

```
` AggregateRequest("@sale_amount:[10000, inf]") .limit(0, 10) .group_by("@state", r.count()) `
```

Will only group by the states found in the first 10 results of the query *@sale_amount:[10000, inf]*. On the other hand,

```
` AggregateRequest("@sale_amount:[10000, inf]") .limit(0, 1000) .group_by("@state", r.count() .limit(0, 10)) `
```

Will group all the results matching the query, but only return the first 10 groups.

If you only wish to return a *top-N* style query, consider using *sort_by()* instead.

Parameters

- **offset** (*int*)
- **num** (*int*)

Return type

AggregateRequest

load(**fields*)

Indicate the fields to be returned in the response. These fields are returned in addition to any others implicitly specified.

Parameters

- **fields:** If fields not specified, all the fields will be loaded.

Otherwise, fields should be given in the format of *@field*.

Parameters**fields** (*str*)**Return type***AggregateRequest***scorer**(*scorer*)

Use a different scoring function to evaluate document relevance. Default is *TFIDF*.

Parameters**scorer** (*str*) – The scoring function to use (e.g. *TFIDF.DOCNORM* or *BM25*)**Return type***AggregateRequest***sort_by**(**fields*, ***kwargs*)

Indicate how the results should be sorted. This can also be used for *top-N* style queries

Parameters

- **fields:** The fields by which to sort. This can be either a single field or a list of fields. If you wish to specify order, you can use the *Asc* or *Desc* wrapper classes.
- **max:** Maximum number of results to return. This can be used instead of *LIMIT* and is also faster.

Example of sorting by *foo* ascending and *bar* descending:

```
` sort_by(Asc("@foo"), Desc("@bar")) `
```

Return the top 10 customers:

```
` AggregateRequest().group_by("@customer", r.sum("@paid").alias(FIELDNAME))  
.sort_by(Desc("@paid"), max=10) `
```

Parameters**fields** (*str*)**Return type***AggregateRequest***with_schema**()

If set, the *schema* property will contain a list of [*field*, *type*] entries in the result object.

Return type*AggregateRequest***property params:** Dict[*str*, Any]

Return the parameters for the aggregation.

Returns

The parameters for the aggregation.

Return typeDict[*str*, Any]

2.2.5 Filter

FilterExpression

class FilterExpression(*_filter=None, operator=None, left=None, right=None*)

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.

```
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:

```
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.

Parameters

- **_filter** (*str* / *None*)
- **operator** (*FilterOperator* / *None*)
- **left** (*FilterExpression* / *None*)
- **right** (*FilterExpression* / *None*)

Tag

class `Tag`(*field*)

A Tag filter can be applied to Tag fields

Parameters

field (*str*)

__eq__(*other*)

Create a Tag equality filter expression.

Parameters

other (*Union[List[str], str]*) – The tag(s) to filter on.

Return type

`FilterExpression`

```
from redisvl.query.filter import Tag

f = Tag("brand") == "nike"
```

__ne__(*other*)

Create a Tag inequality filter expression.

Parameters

other (*Union[List[str], str]*) – The tag(s) to filter on.

Return type

`FilterExpression`

```
from redisvl.query.filter import Tag

f = Tag("brand") != "nike"
```

__str__()

Return the Redis Query string for the Tag filter

Return type

`str`

Text

class `Text`(*field*)

A Text is a FilterField representing a text field in a Redis index.

Parameters

field (*str*)

__eq__(*other*)

Create a Text equality filter expression. These expressions yield filters that enforce an exact match on the supplied term(s).

Parameters

other (*str*) – The text value to filter on.

Return type

`FilterExpression`

```
from redisvl.query.filter import Text

f = Text("job") == "engineer"
```

__mod__(*other*)

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).

Parameters

other (*str*) – The text value to filter on.

Return type

FilterExpression

```
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
```

__ne__(*other*)

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.

Parameters

other (*str*) – The text value to filter on.

Return type

FilterExpression

```
from redisvl.query.filter import Text

f = Text("job") != "engineer"
```

__str__()

Return the Redis Query string for the Text filter

Return type

str

Num**class Num**(*field*)

A Num is a FilterField representing a numeric field in a Redis index.

Parameters

field (*str*)

__eq__(*other*)

Create a Numeric equality filter expression.

Parameters

other (*int*) – The value to filter on.

Return type

FilterExpression

```
from redisvl.query.filter import Num
f = Num("zipcode") == 90210
```

__ge__(*other*)

Create a Numeric greater than or equal to filter expression.

Parameters

other (*int*) – The value to filter on.

Return type

FilterExpression

```
from redisvl.query.filter import Num

f = Num("age") >= 18
```

__gt__(*other*)

Create a Numeric greater than filter expression.

Parameters

other (*int*) – The value to filter on.

Return type

FilterExpression

```
from redisvl.query.filter import Num

f = Num("age") > 18
```

__le__(*other*)

Create a Numeric less than or equal to filter expression.

Parameters

other (*int*) – The value to filter on.

Return type

FilterExpression

```
from redisvl.query.filter import Num

f = Num("age") <= 18
```

__lt__(*other*)

Create a Numeric less than filter expression.

Parameters

other (*int*) – The value to filter on.

Return type

FilterExpression

```
from redisvl.query.filter import Num

f = Num("age") < 18
```

__ne__(*other*)

Create a Numeric inequality filter expression.

Parameters**other** (*int*) – The value to filter on.**Return type**

FilterExpression

```
from redisvl.query.filter import Num

f = Num("zipcode") != 90210
```

__str__()

Return the Redis Query string for the Numeric filter

Return type

str

between(*start*, *end*, *inclusive*='both')

Operator for searching values between two numeric values.

Parameters

- **start** (*int*)
- **end** (*int*)
- **inclusive** (*str*)

Return type

FilterExpression

Geo**class Geo**(*field*)

A Geo is a FilterField representing a geographic (lat/lon) field in a Redis index.

Parameters**field** (*str*)**__eq__**(*other*)

Create a geographic filter within a specified GeoRadius.

Parameters**other** (*GeoRadius*) – The geographic spec to filter on.**Return type**

FilterExpression

```
from redisvl.query.filter import Geo, GeoRadius

f = Geo("location") == GeoRadius(-122.4194, 37.7749, 1, unit="m")
```

__ne__(*other*)

Create a geographic filter outside of a specified GeoRadius.

Parameters**other** (*GeoRadius*) – The geographic spec to filter on.**Return type**

FilterExpression

```
from redisvl.query.filter import Geo, GeoRadius

f = Geo("location") != GeoRadius(-122.4194, 37.7749, 1, unit="m")
```

__str__()

Return the Redis Query string for the Geo filter

Return type

str

GeoRadius

class GeoRadius(*longitude, latitude, radius=1, unit='km'*)

A GeoRadius is a GeoSpec representing a geographic radius.

Create a GeoRadius specification (GeoSpec)

Parameters

- **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”.

__init__(*longitude, latitude, radius=1, unit='km'*)

Create a GeoRadius specification (GeoSpec)

Parameters

- **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”.

2.2.6 Vectorizers

HFTextVectorizer

class HFTextVectorizer(*model='sentence-transformers/all-mpnet-base-v2', dtype='float32', cache=None, *, dims=None*)

Bases: BaseVectorizer

The HFTextVectorizer class leverages Hugging Face’s Sentence Transformers for generating vector embeddings from text input.

This vectorizer is particularly useful in scenarios where advanced natural language processing and understanding are required, and ideal for running on your own hardware without usage fees.

You can optionally enable caching to improve performance when generating embeddings for repeated text inputs.

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 embedding needs.

Requirements:

- The *sentence-transformers* library must be installed with pip.

```
# Basic usage
vectorizer = HFTextVectorizer(model="sentence-transformers/all-mpnet-base-v2")
embedding = vectorizer.embed("Hello, world!")

# With caching enabled
from redisvl.extensions.cache.embeddings import EmbeddingsCache
cache = EmbeddingsCache(name="my_embeddings_cache")

vectorizer = HFTextVectorizer(
    model="sentence-transformers/all-mpnet-base-v2",
    cache=cache
)

# First call will compute and cache the embedding
embedding1 = vectorizer.embed("Hello, world!")

# Second call will retrieve from cache
embedding2 = vectorizer.embed("Hello, world!")

# Batch processing
embeddings = vectorizer.embed_many(
    ["Hello, world!", "How are you?"],
    batch_size=2
)
```

Initialize the Hugging Face text vectorizer.

Parameters

- **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'.
- **dtype** (*str*) – the default datatype to use when embedding text as byte arrays. Used when setting *as_buffer=True* in calls to *embed()* and *embed_many()*. Defaults to 'float32'.
- **cache** (*Optional[EmbeddingsCache]*) – Optional *EmbeddingsCache* instance to cache embeddings for better performance with repeated texts. Defaults to *None*.
- ****kwargs** – Additional parameters to pass to the *SentenceTransformer* constructor.
- **dims** (*Annotated[int | None, FieldInfo(annotation=NoneType, required=True, metadata=[Strict(strict=True), Gt(gt=0)])]*)

Raises

- **ImportError** – If the *sentence-transformers* library is not installed.
- **ValueError** – If there is an error setting the embedding model dimensions.
- **ValueError** – If an invalid dtype is provided.

model_post_init(*context*, /)

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** (*BaseModel*) – The BaseModel instance.
- **context** (*Any*) – The context.

Return type

None

model_config: `ClassVar[ConfigDict] = {'arbitrary_types_allowed': True}`

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

property type: `str`

Return the type of vectorizer.

OpenAITextVectorizer

class OpenAITextVectorizer(*model='text-embedding-ada-002', api_config=None, dtype='float32', cache=None, *, dims=None*)

Bases: `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.

You can optionally enable caching to improve performance when generating embeddings for repeated text inputs.

```
# Basic usage with OpenAI embeddings
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!")

# With caching enabled
from redisvl.extensions.cache.embeddings import EmbeddingsCache
cache = EmbeddingsCache(name="openai_embeddings_cache")

vectorizer = OpenAITextVectorizer(
    model="text-embedding-ada-002",
    api_config={"api_key": "your_api_key"},
    cache=cache
)

# First call will compute and cache the embedding
```

(continues on next page)

(continued from previous page)

```

embedding1 = vectorizer.embed("Hello, world!")

# Second call will retrieve from cache
embedding2 = vectorizer.embed("Hello, world!")

# Asynchronous batch embedding of multiple texts
embeddings = await vectorizer.aembed_many(
    ["Hello, world!", "How are you?"],
    batch_size=2
)

```

Initialize the OpenAI vectorizer.

Parameters

- **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.
- **dtype** (*str*) – the default datatype to use when embedding text as byte arrays. Used when setting *as_buffer=True* in calls to *embed()* and *embed_many()*. Defaults to ‘float32’.
- **cache** (*Optional[EmbeddingsCache]*) – Optional *EmbeddingsCache* instance to cache embeddings for better performance with repeated texts. Defaults to None.
- **dims** (*Annotated[int | None, FieldInfo(annotation=NoneType, required=True, metadata=[Strict(strict=True), Gt(gt=0)])]*)

Raises

- **ImportError** – If the *openai* library is not installed.
- **ValueError** – If the OpenAI API key is not provided.
- **ValueError** – If an invalid *dtype* is provided.

model_config: `ClassVar[ConfigDict] = {'arbitrary_types_allowed': True}`

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

property type: `str`

Return the type of vectorizer.

AzureOpenAITextVectorizer

```

class AzureOpenAITextVectorizer(model='text-embedding-ada-002', api_config=None, dtype='float32',
                                cache=None, *, dims=None)

```

Bases: `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.

You can optionally enable caching to improve performance when generating embeddings for repeated text inputs.

```
# Basic usage
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!")

# With caching enabled
from redisvl.extensions.cache.embeddings import EmbeddingsCache
cache = EmbeddingsCache(name="azureopenai_embeddings_cache")

vectorizer = AzureOpenAITextVectorizer(
    model="text-embedding-ada-002",
    api_config={
        "api_key": "your_api_key",
        "api_version": "your_api_version",
        "azure_endpoint": "your_azure_endpoint",
    },
    cache=cache
)

# First call will compute and cache the embedding
embedding1 = vectorizer.embed("Hello, world!")

# Second call will retrieve from cache
embedding2 = vectorizer.embed("Hello, world!")

# Asynchronous batch embedding of multiple texts
embeddings = await vectorizer.aembed_many(
    ["Hello, world!", "How are you?"],
    batch_size=2
)
```

Initialize the AzureOpenAI vectorizer.

Parameters

- **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.
- **dtype** (*str*) – the default datatype to use when embedding text as byte arrays. Used when setting *as_buffer=True* in calls to *embed()* and *embed_many()*. Defaults to ‘float32’.
- **cache** (*Optional[EmbeddingsCache]*) – Optional *EmbeddingsCache* instance to cache

embeddings for better performance with repeated texts. Defaults to None.

- **dims** (*Annotated[int | None, FieldInfo(annotation=NoneType, required=True, metadata=[Strict(strict=True), Gt(gt=0)])]*)

Raises

- **ImportError** – If the openai library is not installed.
- **ValueError** – If the AzureOpenAI API key, version, or endpoint are not provided.
- **ValueError** – If an invalid dtype is provided.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True}

Configuration for the model, should be a dictionary conforming to [ConfigDict][pydantic.config.ConfigDict].

property type: str

Return the type of vectorizer.

VertexAITextVectorizer

```
class VertexAITextVectorizer(model='textembedding-gecko', api_config=None, dtype='float32',
                             cache=None, *, dims=None)
```

Bases: 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*.

You can optionally enable caching to improve performance when generating embeddings for repeated text inputs.

```
# Basic usage
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!")

# With caching enabled
from redisvl.extensions.cache.embeddings import EmbeddingsCache
cache = EmbeddingsCache(name="vertexai_embeddings_cache")

vectorizer = VertexAITextVectorizer(
    model="textembedding-gecko",
    api_config={
        "project_id": "your_gcp_project_id",
        "location": "your_gcp_location",
    },
```

(continues on next page)

(continued from previous page)

```

    cache=cache
)

# First call will compute and cache the embedding
embedding1 = vectorizer.embed("Hello, world!")

# Second call will retrieve from cache
embedding2 = vectorizer.embed("Hello, world!")

# Batch embedding of multiple texts
embeddings = vectorizer.embed_many(
    ["Hello, world!", "Goodbye, world!"],
    batch_size=2
)

```

Initialize the VertexAI vectorizer.

Parameters

- **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.
- **dtype** (*str*) – the default datatype to use when embedding text as byte arrays. Used when setting *as_buffer=True* in calls to *embed()* and *embed_many()*. Defaults to ‘float32’.
- **cache** (*Optional[EmbeddingsCache]*) – Optional *EmbeddingsCache* instance to cache embeddings for better performance with repeated texts. Defaults to None.
- **dims** (*Annotated[int | None, FieldInfo(annotation=NoneType, required=True, metadata=[Strict(strict=True), Gt(gt=0)])]*)

Raises

- **ImportError** – If the google-cloud-aiplatform library is not installed.
- **ValueError** – If the API key is not provided.
- **ValueError** – If an invalid dtype is provided.

model_config: *ClassVar[ConfigDict]* = {'arbitrary_types_allowed': True}

Configuration for the model, should be a dictionary conforming to *[ConfigDict][pydantic.config.ConfigDict]*.

property type: *str*

Return the type of vectorizer.

CohereTextVectorizer

```
class CohereTextVectorizer(model='embed-english-v3.0', api_config=None, dtype='float32', cache=None, *,
                           dims=None)
```

Bases: 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.

You can optionally enable caching to improve performance when generating embeddings for repeated text inputs.

```
from redisvl.utils.vectorize import CohereTextVectorizer

# Basic usage
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 = vectorizer.embed_many(
    texts=["your document text", "more document text"],
    input_type="search_document"
)

# With caching enabled
from redisvl.extensions.cache.embeddings import EmbeddingsCache
cache = EmbeddingsCache(name="cohere_embeddings_cache")

vectorizer = CohereTextVectorizer(
    model="embed-english-v3.0",
    api_config={"api_key": "your-cohere-api-key"},
    cache=cache
)

# First call will compute and cache the embedding
embedding1 = vectorizer.embed(
    text="your input query text here",
    input_type="search_query"
)

# Second call will retrieve from cache
embedding2 = vectorizer.embed(
    text="your input query text here",
    input_type="search_query"
```

(continues on next page)

(continued from previous page)

)

Initialize the Cohere vectorizer.

Visit <https://cohere.ai/embed> to learn about embeddings.

Parameters

- **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.
- **dtype** (*str*) – the default datatype to use when embedding text as byte arrays. Used when setting *as_buffer=True* in calls to *embed()* and *embed_many()*. ‘float32’ will use Cohere’s float embeddings, ‘int8’ and ‘uint8’ will map to Cohere’s corresponding embedding types. Defaults to ‘float32’.
- **cache** (*Optional[EmbeddingsCache]*) – Optional EmbeddingsCache instance to cache embeddings for better performance with repeated texts. Defaults to None.
- **dims** (*Annotated[int | None, FieldInfo(annotation=NoneType, required=True, metadata=[Strict(strict=True), Gt(gt=0)])]*)

Raises

- **ImportError** – If the cohere library is not installed.
- **ValueError** – If the API key is not provided.
- **ValueError** – If an invalid dtype is provided.

model_config: `ClassVar[ConfigDict] = {'arbitrary_types_allowed': True}`

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

property type: `str`

Return the type of vectorizer.

BedrockTextVectorizer

class BedrockTextVectorizer(*model='amazon.titan-embed-text-v2:0', api_config=None, dtype='float32', cache=None, *, dims=None*)

Bases: `BaseVectorizer`

The AmazonBedrockTextVectorizer class utilizes Amazon Bedrock’s API to generate embeddings for text data.

This vectorizer is designed to interact with Amazon Bedrock API, requiring AWS credentials for authentication. The credentials can be provided directly in the *api_config* dictionary or through environment variables: - AWS_ACCESS_KEY_ID - AWS_SECRET_ACCESS_KEY - AWS_REGION (defaults to us-east-1)

The vectorizer supports synchronous operations with batch processing and preprocessing capabilities.

You can optionally enable caching to improve performance when generating embeddings for repeated text inputs.

```
# Basic usage with explicit credentials
vectorizer = AmazonBedrockTextVectorizer(
    model="amazon.titan-embed-text-v2:0",
    api_config={
```

(continues on next page)

(continued from previous page)

```

        "aws_access_key_id": "your_access_key",
        "aws_secret_access_key": "your_secret_key",
        "aws_region": "us-east-1"
    }
)

# With environment variables and caching
from redisvl.extensions.cache.embeddings import EmbeddingsCache
cache = EmbeddingsCache(name="bedrock_embeddings_cache")

vectorizer = AmazonBedrockTextVectorizer(
    model="amazon.titan-embed-text-v2:0",
    cache=cache
)

# First call will compute and cache the embedding
embedding1 = vectorizer.embed("Hello, world!")

# Second call will retrieve from cache
embedding2 = vectorizer.embed("Hello, world!")

# Generate batch embeddings
embeddings = vectorizer.embed_many(["Hello", "World"], batch_size=2)

```

Initialize the AWS Bedrock Vectorizer.

Parameters

- **model** (*str*) – The Bedrock model ID to use. Defaults to `amazon.titan-embed-text-v2:0`
- **api_config** (*Optional[Dict[str, str]]*) – AWS credentials and config. Can include: `aws_access_key_id`, `aws_secret_access_key`, `aws_region`. If not provided, will use environment variables.
- **dtype** (*str*) – the default datatype to use when embedding text as byte arrays. Used when setting `as_buffer=True` in calls to `embed()` and `embed_many()`. Defaults to `'float32'`.
- **cache** (*Optional[EmbeddingsCache]*) – Optional `EmbeddingsCache` instance to cache embeddings for better performance with repeated texts. Defaults to `None`.
- **dims** (*Annotated[int | None, FieldInfo(annotation=NoneType, required=True, metadata=[Strict(strict=True), Gt(gt=0)])]*)

Raises

- **ValueError** – If credentials are not provided in config or environment.
- **ImportError** – If `boto3` is not installed.
- **ValueError** – If an invalid dtype is provided.

model_config: `ClassVar[ConfigDict] = {'arbitrary_types_allowed': True}`

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

property type: `str`

Return the type of vectorizer.

CustomTextVectorizer

```
class CustomTextVectorizer(embed, embed_many=None, aembed=None, aembed_many=None,
                           dtype='float32', cache=None)
```

Bases: BaseVectorizer

The CustomTextVectorizer class wraps user-defined embedding 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 synchronous embedding is required to satisfy the 'embed()' method.

You can optionally enable caching to improve performance when generating embeddings for repeated text inputs.

```
# Basic usage with a custom embedding function
vectorizer = CustomTextVectorizer(
    embed = my_vectorizer.generate_embedding
)
embedding = vectorizer.embed("Hello, world!")

# With caching enabled
from redisvl.extensions.cache.embeddings import EmbeddingsCache
cache = EmbeddingsCache(name="my_embeddings_cache")

vectorizer = CustomTextVectorizer(
    embed=my_vectorizer.generate_embedding,
    cache=cache
)

# First call will compute and cache the embedding
embedding1 = vectorizer.embed("Hello, world!")

# Second call will retrieve from cache
embedding2 = vectorizer.embed("Hello, world!")

# Asynchronous batch embedding of multiple texts
embeddings = await vectorizer.aembed_many(
    ["Hello, world!", "How are you?"],
    batch_size=2
)
```

Initialize the Custom vectorizer.

Parameters

- **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 asynchronous Callable function that accepts a string object and returns a lists of floats. Defaults to None.
- **aembed_many** (*Optional[Callable]*) – an asynchronous Callable function that accepts a list of string objects and returns a list containing lists of floats. Defaults to None.

- **dtype** (*str*) – the default datatype to use when embedding text as byte arrays. Used when setting *as_buffer=True* in calls to *embed()* and *embed_many()*. Defaults to 'float32'.
- **cache** (*Optional[EmbeddingsCache]*) – Optional *EmbeddingsCache* instance to cache embeddings for better performance with repeated texts. Defaults to *None*.

Raises

ValueError – if embedding validation fails.

model_config: `ClassVar[ConfigDict] = {'arbitrary_types_allowed': True}`

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

property type: `str`

Return the type of vectorizer.

VoyageAITextVectorizer

class VoyageAITextVectorizer(*model='voyage-large-2', api_config=None, dtype='float32', cache=None, *, dims=None*)

Bases: *BaseVectorizer*

The *VoyageAITextVectorizer* class utilizes VoyageAI's API to generate embeddings for text data.

This vectorizer is designed to interact with VoyageAI's /embed API, requiring an API key for authentication. The key can be provided directly in the *api_config* dictionary or through the *VOYAGE_API_KEY* environment variable. User must obtain an API key from VoyageAI's website (<https://dash.voyageai.com/>). Additionally, the *voyageai* python client must be installed with *pip install voyageai*.

The vectorizer supports both synchronous and asynchronous operations, allows for batch processing of texts and flexibility in handling preprocessing tasks.

You can optionally enable caching to improve performance when generating embeddings for repeated text inputs.

```
from redisvl.utils.vectorize import VoyageAITextVectorizer

# Basic usage
vectorizer = VoyageAITextVectorizer(
    model="voyage-large-2",
    api_config={"api_key": "your-voyageai-api-key"} # OR set VOYAGE_API_KEY in your
    ↪ env
)
query_embedding = vectorizer.embed(
    text="your input query text here",
    input_type="query"
)
doc_embeddings = vectorizer.embed_many(
    texts=["your document text", "more document text"],
    input_type="document"
)

# With caching enabled
from redisvl.extensions.cache.embeddings import EmbeddingsCache
cache = EmbeddingsCache(name="voyageai_embeddings_cache")

vectorizer = VoyageAITextVectorizer(
```

(continues on next page)

(continued from previous page)

```

model="voyage-large-2",
api_config={"api_key": "your-voyageai-api-key"},
cache=cache
)

# First call will compute and cache the embedding
embedding1 = vectorizer.embed(
    text="your input query text here",
    input_type="query"
)

# Second call will retrieve from cache
embedding2 = vectorizer.embed(
    text="your input query text here",
    input_type="query"
)

```

Initialize the VoyageAI vectorizer.

Visit <https://docs.voyageai.com/docs/embeddings> to learn about embeddings and check the available models.

Parameters

- **model** (*str*) – Model to use for embedding. Defaults to “voyage-large-2”.
- **api_config** (*Optional[Dict]*, *optional*) – Dictionary containing the API key. Defaults to None.
- **dtype** (*str*) – the default datatype to use when embedding text as byte arrays. Used when setting *as_buffer=True* in calls to *embed()* and *embed_many()*. Defaults to ‘float32’.
- **cache** (*Optional[EmbeddingsCache]*) – Optional EmbeddingsCache instance to cache embeddings for better performance with repeated texts. Defaults to None.
- **dims** (*Annotated[int | None, FieldInfo(annotation=NoneType, required=True, metadata=[Strict(strict=True), Gt(gt=0)])]*)

Raises

- **ImportError** – If the voyageai library is not installed.
- **ValueError** – If the API key is not provided.

model_config: `ClassVar[ConfigDict] = {'arbitrary_types_allowed': True}`

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

property type: `str`

Return the type of vectorizer.

2.2.7 Rerankers

CohereReranker

```
class CohereReranker(model='rerank-english-v3.0', rank_by=None, limit=5, return_score=True,
                      api_config=None)
```

Bases: 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*.

```
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"}
    ]
)
```

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.

```
async arank(query, docs, **kwargs)
```

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

The reranked list of documents and optionally associated scores.

Return type

`Union[Tuple[Union[List[Dict[str, Any]], List[str]], float], List[Dict[str, Any]]]`

model_post_init(*context*, /)

This function is meant to behave like a `BaseModel` method to initialise private attributes.

It takes *context* as an argument since that’s what `pydantic-core` passes when calling it.

Parameters

- **self** (*BaseModel*) – The `BaseModel` instance.
- **context** (*Any*) – The context.

Return type

`None`

rank(*query*, *docs*, ***kwargs*)

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

The reranked list of documents and optionally associated scores.

Return type

`Union[Tuple[Union[List[Dict[str, Any]], List[str]], float], List[Dict[str, Any]]]`

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

HFCrossEncoderReranker

```
class HFCrossEncoderReranker(model='cross-encoder/ms-marco-MiniLM-L-6-v2', limit=3,
                             return_score=True, *, rank_by=None)
```

Bases: `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.

```

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"}
    ]
)

```

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.
- **rank_by** (*List[str] | None*)

async arank(*query, docs, **kwargs*)

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

The reranked list of documents and optionally associated scores.

Return type

`Union[Tuple[List[Dict[str, Any]], List[float]], List[Dict[str, Any]]]`

model_post_init(*context, /*)

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** (*BaseModel*) – The BaseModel instance.
- **context** (*Any*) – The context.

Return type

`None`

rank(*query*, *docs*, ***kwargs*)

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

The reranked list of documents and optionally associated scores.

Return type

`Union[Tuple[List[Dict[str, Any]], List[float]], List[Dict[str, Any]]]`

model_config: `ClassVar[ConfigDict] = {}`

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

VoyageAIReranker

class VoyageAIReranker(*model*, *rank_by=None*, *limit=5*, *return_score=True*, *api_config=None*)

Bases: `BaseReranker`

The VoyageAIReranker class uses VoyageAI's API to rerank documents based on an input query.

This reranker is designed to interact with VoyageAI's /rerank API, requiring an API key for authentication. The key can be provided directly in the *api_config* dictionary or through the `VOYAGE_API_KEY` environment variable. User must obtain an API key from VoyageAI's website (<https://dash.voyageai.com/>). Additionally, the *voyageai* python client must be installed with `pip install voyageai`.

```
from redisvl.utils.rerank import VoyageAIReranker

# set up the VoyageAI reranker with some configuration
reranker = VoyageAIReranker(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"}
    ]
)
```

Initialize the VoyageAIReranker with specified model, ranking criteria, and API configuration.

Parameters

- **model** (*str*) – The identifier for the VoyageAI model used for reranking.
- **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 voyageai library is not installed.
- **ValueError** – If the API key is not provided.

async arank(*query*, *docs*, ***kwargs*)

Rerank documents based on the provided query using the VoyageAI 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

The reranked list of documents and optionally associated scores.

Return type

Union[Tuple[Union[List[Dict[str, Any]], List[str]], float], List[Dict[str, Any]]]

model_post_init(*context*, /)

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** (*BaseModel*) – The BaseModel instance.
- **context** (*Any*) – The context.

Return type

None

rank(*query*, *docs*, ***kwargs*)

Rerank documents based on the provided query using the VoyageAI 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

The reranked list of documents and optionally associated scores.

Return type

Union[Tuple[Union[List[Dict[str, Any]], List[str]], float], List[Dict[str, Any]]]

```
model_config: ClassVar[ConfigDict] = {}
```

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

2.2.8 LLM Cache

SemanticCache

```
class SemanticCache(name='llmcache', distance_threshold=0.1, ttl=None, vectorizer=None,
                    filterable_fields=None, redis_client=None, redis_url='redis://localhost:6379',
                    connection_kwargs={}, overwrite=False, **kwargs)
```

Bases: `BaseLLMCache`

Semantic Cache for Large Language Models.

Semantic Cache for Large Language Models.

Parameters

- **name** (*str*, *optional*) – The name of the semantic cache search index. Defaults to “llm-cache”.
- **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.

```
async acheck(prompt=None, vector=None, num_results=1, return_fields=None, filter_expression=None,
             distance_threshold=None)
```

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.

Parameters

- **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

A list of dicts containing the requested
return fields for each similar cached response.

Return type

List[Dict[str, Any]]

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.

```
response = await cache.acheck(
    prompt="What is the capital city of France?"
)
```

async aclear()

Async clear the cache of all keys.

Return type

None

async adelete()

Async delete the cache and its index entirely.

Return type

None

async adisconnect()

Asynchronously disconnect from Redis and search index.

Closes all Redis connections and index connections.

async adrop(ids=None, keys=None)

Async drop specific entries from the cache by ID or Redis key.

Parameters

- **ids** (*Optional[List[str]]*) – List of entry IDs to remove from the cache. Entry IDs are the unique identifiers without the cache prefix.

- **keys** (*Optional[List[str]]*) – List of full Redis keys to remove from the cache. Keys are the complete Redis keys including the cache prefix.

Return type

None

Note

At least one of ids or keys must be provided.

Raises

ValueError – If neither ids nor keys is provided.

Parameters

- **ids** (*List[str] | None*)
- **keys** (*List[str] | None*)

Return type

None

async aexpire(*key, ttl=None*)

Asynchronously set or refresh the expiration time for a key in the cache.

Parameters

- **key** (*str*) – The Redis key to set the expiration on.
- **ttl** (*Optional[int], optional*) – The time-to-live in seconds. If None, uses the default TTL configured for this cache instance. Defaults to None.

Return type

None

Note

If neither the provided TTL nor the default TTL is set (both are None), this method will have no effect.

async astore(*prompt, response, vector=None, metadata=None, filters=None, ttl=None*)

Async stores the specified key-value pair in the cache along with metadata.

Parameters

- **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

The Redis key for the entries added to the semantic cache.

Return type

str

Raises

- **ValueError** – If neither prompt nor vector is specified.
- **ValueError** – if vector has incorrect dimensions.
- **TypeError** – If provided metadata is not a dictionary.

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

async aupdate(key, **kwargs)

Async update specific fields within an existing cache entry. If no fields are passed, then only the document TTL is refreshed.

Parameters

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. –

Return type

None

```
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"}
)
```

check(prompt=None, vector=None, num_results=1, return_fields=None, filter_expression=None, distance_threshold=None)

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.

Parameters

- **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

A list of dicts containing the requested
return fields for each similar cached response.

Return type

List[Dict[str, Any]]

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.

```
response = cache.check(  
    prompt="What is the capital city of France?"  
)
```

clear()

Clear the cache of all keys.

Return type

None

delete()

Delete the cache and its index entirely.

Return type

None

disconnect()

Disconnect from Redis and search index.

Closes all Redis connections and index connections.

drop(ids=None, keys=None)

Drop specific entries from the cache by ID or Redis key.

Parameters

- **ids** (*Optional[List[str]]*) – List of entry IDs to remove from the cache. Entry IDs are the unique identifiers without the cache prefix.
- **keys** (*Optional[List[str]]*) – List of full Redis keys to remove from the cache. Keys are the complete Redis keys including the cache prefix.

Return type

None

Note

At least one of ids or keys must be provided.

Raises

ValueError – If neither ids nor keys is provided.

Parameters

- **ids** (*List[str] | None*)
- **keys** (*List[str] | None*)

Return type

None

expire(*key, ttl=None*)

Set or refresh the expiration time for a key in the cache.

Parameters

- **key** (*str*) – The Redis key to set the expiration on.
- **ttl** (*Optional[int], optional*) – The time-to-live in seconds. If None, uses the default TTL configured for this cache instance. Defaults to None.

Return type

None

Note

If neither the provided TTL nor the default TTL is set (both are None), this method will have no effect.

set_threshold(*distance_threshold*)

Sets the semantic distance threshold for the cache.

Parameters

distance_threshold (*float*) – The semantic distance threshold for the cache.

Raises

ValueError – If the threshold is not between 0 and 1.

Return type

None

set_ttl(*ttl=None*)

Set the default TTL, in seconds, for entries in the cache.

Parameters

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.

Return type

None

store(*prompt, response, vector=None, metadata=None, filters=None, ttl=None*)

Stores the specified key-value pair in the cache along with metadata.

Parameters

- **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

The Redis key for the entries added to the semantic cache.

Return type

str

Raises

- **ValueError** – If neither prompt nor vector is specified.
- **ValueError** – if vector has incorrect dimensions.
- **TypeError** – If provided metadata is not a dictionary.

```
key = cache.store(  
    prompt="What is the capital city of France?",  
    response="Paris",  
    metadata={"city": "Paris", "country": "France"}  
)
```

update(*key, **kwargs*)

Update specific fields within an existing cache entry. If no fields are passed, then only the document TTL is refreshed.

Parameters

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. –

Return type

None

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

property aindex: [*AsyncSearchIndex*](#) | None

The underlying AsyncSearchIndex for the cache.

Returns

The async search index.

Return type

[*AsyncSearchIndex*](#)

property distance_threshold: float

The semantic distance threshold for the cache.

Returns

The semantic distance threshold.

Return type

float

property index: [SearchIndex](#)

The underlying SearchIndex for the cache.

Returns

The search index.

Return type

[SearchIndex](#)

property ttl: int | None

The default TTL, in seconds, for entries in the cache.

2.2.9 Embeddings Cache

EmbeddingsCache

```
class EmbeddingsCache(name='embedcache', ttl=None, redis_client=None, async_redis_client=None,
                      redis_url='redis://localhost:6379', connection_kwargs={})
```

Bases: BaseCache

Embeddings Cache for storing embedding vectors with exact key matching.

Initialize an embeddings cache.

Parameters

- **name** (*str*) – The name of the cache. Defaults to “embedcache”.
- **ttl** (*Optional[int]*) – The time-to-live for cached embeddings. Defaults to None.
- **redis_client** (*Optional[SyncRedisClient]*) – Redis client instance. Defaults to None.
- **redis_url** (*str*) – Redis URL for connection. Defaults to “redis://localhost:6379”.
- **connection_kwargs** (*Dict[str, Any]*) – Redis connection arguments. Defaults to {}.
- **async_redis_client** (*Redis | RedisCluster | None*)

Raises

ValueError – If vector dimensions are invalid

```
cache = EmbeddingsCache(
    name="my_embeddings_cache",
    ttl=3600, # 1 hour
    redis_url="redis://localhost:6379"
)
```

async aclear()

Async clear the cache of all keys.

Return type

None

async adisconnect()

Async disconnect from Redis.

Return type

None

async adrop(text, model_name)

Async remove an embedding from the cache.

Asynchronously removes an embedding from the cache.

Parameters

- **text** (*str*) – The text input that was embedded.
- **model_name** (*str*) – The name of the embedding model.

Return type

None

```
await cache.adrop(  
    text="What is machine learning?",  
    model_name="text-embedding-ada-002"  
)
```

async adrop_by_key(key)

Async remove an embedding from the cache by its Redis key.

Asynchronously removes an embedding from the cache by its Redis key.

Parameters**key** (*str*) – The full Redis key for the embedding.**Return type**

None

```
await cache.adrop_by_key("embedcache:1234567890abcdef")
```

async aexists(text, model_name)

Async check if an embedding exists.

Asynchronously checks if an embedding exists for the given text and model.

Parameters

- **text** (*str*) – The text input that was embedded.
- **model_name** (*str*) – The name of the embedding model.

Returns

True if the embedding exists in the cache, False otherwise.

Return type

bool

```
if await cache.aexists("What is machine learning?", "text-embedding-ada-002"):  
    print("Embedding is in cache")
```

async aexists_by_key(key)

Async check if an embedding exists for the given Redis key.

Asynchronously checks if an embedding exists for the given Redis key.

Parameters

key (*str*) – The full Redis key for the embedding.

Returns

True if the embedding exists in the cache, False otherwise.

Return type

bool

```
if await cache.aexists_by_key("embedcache:1234567890abcdef"):
    print("Embedding is in cache")
```

async aexpire(key, ttl=None)

Asynchronously set or refresh the expiration time for a key in the cache.

Parameters

- **key** (*str*) – The Redis key to set the expiration on.
- **ttl** (*Optional[int], optional*) – The time-to-live in seconds. If None, uses the default TTL configured for this cache instance. Defaults to None.

Return type

None

Note

If neither the provided TTL nor the default TTL is set (both are None), this method will have no effect.

async aget(text, model_name)

Async get embedding by text and model name.

Asynchronously retrieves a cached embedding for the given text and model name. If found, refreshes the TTL of the entry.

Parameters

- **text** (*str*) – The text input that was embedded.
- **model_name** (*str*) – The name of the embedding model.

Returns

Embedding cache entry or None if not found.

Return type

Optional[Dict[str, Any]]

```
embedding_data = await cache.aget(
    text="What is machine learning?",
    model_name="text-embedding-ada-002"
)
```

async aget_by_key(key)

Async get embedding by its full Redis key.

Asynchronously retrieves a cached embedding for the given Redis key. If found, refreshes the TTL of the entry.

Parameters

key (*str*) – The full Redis key for the embedding.

Returns

Embedding cache entry or None if not found.

Return type

Optional[Dict[str, Any]]

```
embedding_data = await cache.aget_by_key("embedcache:1234567890abcdef")
```

async amdrop(texts, model_name)

Async remove multiple embeddings from the cache by their texts and model name.

Asynchronously removes multiple embeddings in a single operation.

Parameters

- **texts** (*List[str]*) – List of text inputs that were embedded.
- **model_name** (*str*) – The name of the embedding model.

Return type

None

```
# Remove multiple embeddings asynchronously
await cache.amdrop(
    texts=["What is machine learning?", "What is deep learning?"],
    model_name="text-embedding-ada-002"
)
```

async amdrop_by_keys(keys)

Async remove multiple embeddings from the cache by their Redis keys.

Asynchronously removes multiple embeddings in a single operation.

Parameters

keys (*List[str]*) – List of Redis keys to remove.

Return type

None

```
# Remove multiple embeddings asynchronously
await cache.amdrop_by_keys(["embedcache:key1", "embedcache:key2"])
```

async amexists(texts, model_name)

Async check if multiple embeddings exist by their texts and model name.

Asynchronously checks existence of multiple embeddings in a single operation.

Parameters

- **texts** (*List[str]*) – List of text inputs that were embedded.
- **model_name** (*str*) – The name of the embedding model.

Returns

List of boolean values indicating whether each embedding exists.

Return type

List[bool]

```
# Check if multiple embeddings exist asynchronously
exists_results = await cache.amexists(
    texts=["What is machine learning?", "What is deep learning?"],
    model_name="text-embedding-ada-002"
)
```

async amexists_by_keys(keys)

Async check if multiple embeddings exist by their Redis keys.

Asynchronously checks existence of multiple keys in a single operation.

Parameters

keys (List[str]) – List of Redis keys to check.

Returns

List of boolean values indicating whether each key exists. The order matches the input keys order.

Return type

List[bool]

```
# Check if multiple keys exist asynchronously
exists_results = await cache.amexists_by_keys(["embedcache:key1", "embedcache:
key2"])
```

async amget(texts, model_name)

Async get multiple embeddings by their texts and model name.

Asynchronously retrieves multiple cached embeddings in a single operation. If found, refreshes the TTL of each entry.

Parameters

- **texts** (List[str]) – List of text inputs that were embedded.
- **model_name** (str) – The name of the embedding model.

Returns

List of embedding cache entries or None for texts not found.

Return type

List[Optional[Dict[str, Any]]]

```
# Get multiple embeddings asynchronously
embedding_data = await cache.amget(
    texts=["What is machine learning?", "What is deep learning?"],
    model_name="text-embedding-ada-002"
)
```

async amget_by_keys(keys)

Async get multiple embeddings by their Redis keys.

Asynchronously retrieves multiple cached embeddings in a single network roundtrip. If found, refreshes the TTL of each entry.

Parameters

keys (*List[str]*) – List of Redis keys to retrieve.

Returns

List of embedding cache entries or None for keys not found. The order matches the input keys order.

Return type

List[Optional[Dict[str, Any]]]

```
# Get multiple embeddings asynchronously
embedding_data = await cache.amget_by_keys([
    "embedcache:key1",
    "embedcache:key2"
])
```

async amset(*items, ttl=None*)

Async store multiple embeddings in a batch operation.

Each item in the input list should be a dictionary with the following fields: - 'text': The text input that was embedded - 'model_name': The name of the embedding model - 'embedding': The embedding vector - 'metadata': Optional metadata to store with the embedding

Parameters

- **items** (*List[Dict[str, Any]]*) – List of dictionaries, each containing text, model_name, embedding, and optional metadata.
- **ttl** (*int | None*) – Optional TTL override for these entries.

Returns

List of Redis keys where the embeddings were stored.

Return type

List[str]

```
# Store multiple embeddings asynchronously
keys = await cache.amset([
    {
        "text": "What is ML?",
        "model_name": "text-embedding-ada-002",
        "embedding": [0.1, 0.2, 0.3],
        "metadata": {"source": "user"}
    },
    {
        "text": "What is AI?",
        "model_name": "text-embedding-ada-002",
        "embedding": [0.4, 0.5, 0.6],
        "metadata": {"source": "docs"}
    }
])
```

async aset(*text, model_name, embedding, metadata=None, ttl=None*)

Async store an embedding with its text and model name.

Asynchronously stores an embedding with its text and model name.

Parameters

- **text** (*str*) – The text input that was embedded.
- **model_name** (*str*) – The name of the embedding model.
- **embedding** (*List[float]*) – The embedding vector to store.
- **metadata** (*Optional[Dict[str, Any]]*) – Optional metadata to store with the embedding.
- **ttl** (*Optional[int]*) – Optional TTL override for this specific entry.

Returns

The Redis key where the embedding was stored.

Return type

str

```
key = await cache aset(
    text="What is machine learning?",
    model_name="text-embedding-ada-002",
    embedding=[0.1, 0.2, 0.3, ...],
    metadata={"source": "user_query"}
)
```

clear()

Clear the cache of all keys.

Return type

None

disconnect()

Disconnect from Redis.

Return type

None

drop(text, model_name)

Remove an embedding from the cache.

Parameters

- **text** (*str*) – The text input that was embedded.
- **model_name** (*str*) – The name of the embedding model.

Return type

None

```
cache.drop(
    text="What is machine learning?",
    model_name="text-embedding-ada-002"
)
```

drop_by_key(key)

Remove an embedding from the cache by its Redis key.

Parameters

key (*str*) – The full Redis key for the embedding.

Return type

None

```
cache.drop_by_key("embedcache:1234567890abcdef")
```

exists(*text*, *model_name*)

Check if an embedding exists for the given text and model.

Parameters

- **text** (*str*) – The text input that was embedded.
- **model_name** (*str*) – The name of the embedding model.

Returns

True if the embedding exists in the cache, False otherwise.

Return type

bool

```
if cache.exists("What is machine learning?", "text-embedding-ada-002"):
    print("Embedding is in cache")
```

exists_by_key(*key*)

Check if an embedding exists for the given Redis key.

Parameters

key (*str*) – The full Redis key for the embedding.

Returns

True if the embedding exists in the cache, False otherwise.

Return type

bool

```
if cache.exists_by_key("embedcache:1234567890abcdef"):
    print("Embedding is in cache")
```

expire(*key*, *ttl=None*)

Set or refresh the expiration time for a key in the cache.

Parameters

- **key** (*str*) – The Redis key to set the expiration on.
- **ttl** (*Optional[int]*, *optional*) – The time-to-live in seconds. If None, uses the default TTL configured for this cache instance. Defaults to None.

Return type

None

Note

If neither the provided TTL nor the default TTL is set (both are None), this method will have no effect.

get(*text*, *model_name*)

Get embedding by text and model name.

Retrieves a cached embedding for the given text and model name. If found, refreshes the TTL of the entry.

Parameters

- **text** (*str*) – The text input that was embedded.
- **model_name** (*str*) – The name of the embedding model.

Returns

Embedding cache entry or None if not found.

Return type

Optional[Dict[str, Any]]

```
embedding_data = cache.get(
    text="What is machine learning?",
    model_name="text-embedding-ada-002"
)
```

get_by_key(*key*)

Get embedding by its full Redis key.

Retrieves a cached embedding for the given Redis key. If found, refreshes the TTL of the entry.

Parameters

key (*str*) – The full Redis key for the embedding.

Returns

Embedding cache entry or None if not found.

Return type

Optional[Dict[str, Any]]

```
embedding_data = cache.get_by_key("embedcache:1234567890abcdef")
```

mdrop(*texts*, *model_name*)

Remove multiple embeddings from the cache by their texts and model name.

Efficiently removes multiple embeddings in a single operation.

Parameters

- **texts** (*List[str]*) – List of text inputs that were embedded.
- **model_name** (*str*) – The name of the embedding model.

Return type

None

```
# Remove multiple embeddings
cache.mdrop(
    texts=["What is machine learning?", "What is deep learning?"],
    model_name="text-embedding-ada-002"
)
```

mdrop_by_keys(*keys*)

Remove multiple embeddings from the cache by their Redis keys.

Efficiently removes multiple embeddings in a single operation.

Parameters

keys (*List[str]*) – List of Redis keys to remove.

Return type

None

```
# Remove multiple embeddings
cache.mdrop_by_keys(["embedcache:key1", "embedcache:key2"])
```

mexists(*texts*, *model_name*)

Check if multiple embeddings exist by their texts and model name.

Efficiently checks existence of multiple embeddings in a single operation.

Parameters

- **texts** (*List[str]*) – List of text inputs that were embedded.
- **model_name** (*str*) – The name of the embedding model.

Returns

List of boolean values indicating whether each embedding exists.

Return type

List[bool]

```
# Check if multiple embeddings exist
exists_results = cache.mexists(
    texts=["What is machine learning?", "What is deep learning?"],
    model_name="text-embedding-ada-002"
)
```

mexists_by_keys(*keys*)

Check if multiple embeddings exist by their Redis keys.

Efficiently checks existence of multiple keys in a single operation.

Parameters

keys (*List[str]*) – List of Redis keys to check.

Returns

List of boolean values indicating whether each key exists. The order matches the input keys order.

Return type

List[bool]

```
# Check if multiple keys exist
exists_results = cache.mexists_by_keys(["embedcache:key1", "embedcache:key2"])
```

mget(*texts*, *model_name*)

Get multiple embeddings by their texts and model name.

Efficiently retrieves multiple cached embeddings in a single operation. If found, refreshes the TTL of each entry.

Parameters

- **texts** (*List[str]*) – List of text inputs that were embedded.
- **model_name** (*str*) – The name of the embedding model.

Returns

List of embedding cache entries or None for texts not found.

Return type

List[Optional[Dict[str, Any]]]

```
# Get multiple embeddings
embedding_data = cache.mget(
    texts=["What is machine learning?", "What is deep learning?"],
    model_name="text-embedding-ada-002"
)
```

mget_by_keys(keys)

Get multiple embeddings by their Redis keys.

Efficiently retrieves multiple cached embeddings in a single network roundtrip. If found, refreshes the TTL of each entry.

Parameters

keys (*List[str]*) – List of Redis keys to retrieve.

Returns

List of embedding cache entries or None for keys not found. The order matches the input keys order.

Return type

List[Optional[Dict[str, Any]]]

```
# Get multiple embeddings
embedding_data = cache.mget_by_keys([
    "embedcache:key1",
    "embedcache:key2"
])
```

mset(items, ttl=None)

Store multiple embeddings in a batch operation.

Each item in the input list should be a dictionary with the following fields: - 'text': The text input that was embedded - 'model_name': The name of the embedding model - 'embedding': The embedding vector - 'metadata': Optional metadata to store with the embedding

Parameters

- **items** (*List[Dict[str, Any]]*) – List of dictionaries, each containing text, model_name, embedding, and optional metadata.
- **ttl** (*int | None*) – Optional TTL override for these entries.

Returns

List of Redis keys where the embeddings were stored.

Return type

List[str]

```
# Store multiple embeddings
keys = cache.mset([
    {
        "text": "What is ML?",
        "model_name": "text-embedding-ada-002",
        "embedding": [0.1, 0.2, 0.3],
        "metadata": {"source": "user"}
    },
    {
```

(continues on next page)

(continued from previous page)

```

        "text": "What is AI?",
        "model_name": "text-embedding-ada-002",
        "embedding": [0.4, 0.5, 0.6],
        "metadata": {"source": "docs"}
    }
}

```

set(text, model_name, embedding, metadata=None, ttl=None)

Store an embedding with its text and model name.

Parameters

- **text** (*str*) – The text input that was embedded.
- **model_name** (*str*) – The name of the embedding model.
- **embedding** (*List[float]*) – The embedding vector to store.
- **metadata** (*Optional[Dict[str, Any]]*) – Optional metadata to store with the embedding.
- **ttl** (*Optional[int]*) – Optional TTL override for this specific entry.

Returns

The Redis key where the embedding was stored.

Return type

str

```

key = cache.set(
    text="What is machine learning?",
    model_name="text-embedding-ada-002",
    embedding=[0.1, 0.2, 0.3, ...],
    metadata={"source": "user_query"}
)

```

set_ttl(ttl=None)

Set the default TTL, in seconds, for entries in the cache.

Parameters

- **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.

Return type

None

property ttl: int | None

The default TTL, in seconds, for entries in the cache.

2.2.10 LLM Message History

SemanticMessageHistory

```
class SemanticMessageHistory(name, session_tag=None, prefix=None, vectorizer=None,
                             distance_threshold=0.3, redis_client=None, redis_url='redis://localhost:6379',
                             connection_kwargs={}, overwrite=False, **kwargs)
```

Bases: BaseMessageHistory

Initialize message history with index

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

Parameters

- **name** (*str*) – The name of the message history index.
- **session_tag** (*Optional[str]*) – Tag to be added to entries to link to a specific conversation session. Defaults to instance ULID.
- **prefix** (*Optional[str]*) – Prefix for the keys for this message 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 message index. Defaults to false.

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

```
add_message(message, session_tag=None)
```

Insert a single prompt or response into the message history. A timestamp is associated with it so that it can be later sorted in sequential ordering after retrieval.

Parameters

- **message** (*Dict[str, str]*) – The user prompt or LLM response.
- **session_tag** (*Optional[str]*) – Tag to be added to entry to link to a specific conversation session. Defaults to instance ULID.

Return type

None

```
add_messages(messages, session_tag=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.

Parameters

- **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 conversation session. Defaults to instance ULID.

Return type

None

clear()

Clears the message history.

Return type

None

delete()

Clear all message keys and remove the search index.

Return type

None

drop(id=None)

Remove a specific exchange from the message history.

Parameters

id (*Optional[str]*) – The id of the message entry to delete. If None then the last entry is deleted.

Return type

None

get_recent(top_k=5, as_text=False, raw=False, session_tag=None, role=None)

Retrieve the recent message history in sequential order.

Parameters

- **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 of the entries linked to a specific conversation session. Defaults to instance ULID.
- **role** (*Optional[Union[str, List[str]]]*) – Filter messages by role(s). Can be a single role string (“system”, “user”, “llm”, “tool”) or a list of roles. If None, all roles are returned.

Returns**A single string transcription of the session**

or list of strings if as_text is false.

Return type

Union[str, List[str]]

Raises

ValueError – if top_k is not an integer greater than or equal to 0, or if role contains invalid values.

get_relevant(*prompt*, *as_text=False*, *top_k=5*, *fall_back=False*, *session_tag=None*, *raw=False*, *distance_threshold=None*, *role=None*)

Searches the message 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.

Parameters

- **prompt** (*str*) – The message text to search for in message history
- **as_text** (*bool*) – Whether to return the prompts and responses as text
- **JSON.** (*or as*)
- **top_k** (*int*) – The number of previous messages to return. Default is 5.
- **session_tag** (*Optional[str]*) – Tag of the entries linked to a specific conversation session. Defaults to instance ULID.
- **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.
- **role** (*Optional[Union[str, List[str]]]*) – Filter messages by role(s). Can be a single role string (“system”, “user”, “lm”, “tool”) or a list of roles. If None, all roles are returned.

Returns

Either a list of strings, or a list of prompts and responses in JSON containing the most relevant.

Return type

Union[List[str], List[Dict[str,str]]

Raises ValueError: if **top_k** is not an integer greater or equal to 0, or if role contains invalid values.

store(*prompt*, *response*, *session_tag=None*)

Insert a prompt:response pair into the message history. A timestamp is associated with each message so that they can be later sorted in sequential ordering after retrieval.

Parameters

- **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 conversation session. Defaults to instance ULID.

Return type

None

property messages: List[str] | List[Dict[str, str]]

Returns the full message history.

MessageHistory

```
class MessageHistory(name, session_tag=None, prefix=None, redis_client=None,  
                    redis_url='redis://localhost:6379', connection_kwargs={}, **kwargs)
```

Bases: BaseMessageHistory

Initialize message history

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

Parameters

- **name** (*str*) – The name of the message history index.
- **session_tag** (*Optional[str]*) – Tag to be added to entries to link to a specific conversation session. Defaults to instance ULID.
- **prefix** (*Optional[str]*) – Prefix for the keys for this conversation 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 {}.

```
add_message(message, session_tag=None)
```

Insert a single prompt or response into the message history. A timestamp is associated with it so that it can be later sorted in sequential ordering after retrieval.

Parameters

- **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 conversation session. Defaults to instance ULID.

Return type

None

```
add_messages(messages, session_tag=None)
```

Insert a list of prompts and responses into the message history. A timestamp is associated with each so that they can be later sorted in sequential ordering after retrieval.

Parameters

- **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 conversation session. Defaults to instance ULID.

Return type

None

```
clear()
```

Clears the conversation message history.

Return type

None

delete()

Clear all conversation keys and remove the search index.

Return type

None

drop(id=None)

Remove a specific exchange from the conversation history.

Parameters

id (*Optional[str]*) – The id of the message entry to delete. If None then the last entry is deleted.

Return type

None

get_recent(top_k=5, as_text=False, raw=False, session_tag=None, role=None)

Retrieve the recent message history in sequential order.

Parameters

- **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 of the entries linked to a specific conversation session. Defaults to instance ULID.
- **role** (*Optional[Union[str, List[str]]]*) – Filter messages by role(s). Can be a single role string (“system”, “user”, “llm”, “tool”) or a list of roles. If None, all roles are returned.

Returns

A single string transcription of the messages
or list of strings if `as_text` is false.

Return type

Union[str, List[str]]

Raises

ValueError – if `top_k` is not an integer greater than or equal to 0, or if `role` contains invalid values.

store(prompt, response, session_tag=None)

Insert a prompt:response pair into the message history. A timestamp is associated with each exchange so that they can be later sorted in sequential ordering after retrieval.

Parameters

- **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 conversation session. Defaults to instance ULID.

Return type

None

property messages: `List[str] | List[Dict[str, str]]`

Returns the full message history.

2.2.11 Semantic Router

Semantic Router

class SemanticRouter(*name, routes, vectorizer=None, routing_config=None, redis_client=None, redis_url='redis://localhost:6379', overwrite=False, connection_kwargs={}*)

Semantic Router for managing and querying route vectors.

Initialize the SemanticRouter.

Parameters

- **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[SyncRedisClient], 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 {}.

add_route_references(*route_name, references*)

Add a reference(s) to an existing route.

Parameters

- **router_name** (*str*) – The name of the router.
- **references** (*Union[str, List[str]]*) – The reference or list of references to add.
- **route_name** (*str*)

Returns

The list of added references keys.

Return type

`List[str]`

clear()

Flush all routes from the semantic router index.

Return type

`None`

delete()

Delete the semantic router index.

Return type

None

delete_route_references(*route_name*="", *reference_ids*=[], *keys*=[])

Get references for an existing semantic router route.

Parameters

- **Optional** (*keys*) – The name of the router.
- **Optional** – The reference or list of references to delete.
- **Optional** – List of fully qualified keys (prefix:router:reference_id) to delete.
- **route_name** (*str*)
- **reference_ids** (*List[str]*)
- **keys** (*List[str]*)

Returns

Number of objects deleted

Return type

int

classmethod from_dict(*data*, ***kwargs*)

Create a SemanticRouter from a dictionary.

Parameters**data** (*Dict[str, Any]*) – The dictionary containing the semantic router data.**Returns**

The semantic router instance.

Return type*SemanticRouter***Raises****ValueError** – If required data is missing or invalid.

```

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)

```

classmethod from_existing(*name*, *redis_client*=None, *redis_url*='redis://localhost:6379', ***kwargs*)

Return SemanticRouter instance from existing index.

Parameters

- **name** (*str*)
- **redis_client** (*Redis | RedisCluster | None*)
- **redis_url** (*str*)

Return type*SemanticRouter*

classmethod `from_yaml(file_path, **kwargs)`

Create a SemanticRouter from a YAML file.

Parameters

file_path (*str*) – The path to the YAML file.

Returns

The semantic router instance.

Return type

SemanticRouter

Raises

- **ValueError** – If the file path is invalid.
- **FileNotFoundError** – If the file does not exist.

```
from redisvl.extensions.router import SemanticRouter
router = SemanticRouter.from_yaml("router.yaml", redis_url="redis://localhost:
↪6379")
```

get(*route_name*)

Get a route by its name.

Parameters

route_name (*str*) – Name of the route.

Returns

The selected Route object or None if not found.

Return type

Optional[*Route*]

get_route_references(*route_name=""*, *reference_ids=[]*, *keys=[]*)

Get references for an existing route route.

Parameters

- **router_name** (*str*) – The name of the router.
- **references** (*Union[str, List[str]]*) – The reference or list of references to add.
- **route_name** (*str*)
- **reference_ids** (*List[str]*)
- **keys** (*List[str]*)

Returns

Reference objects stored

Return type

List[Dict[str, Any]]

model_post_init(*context*, /)

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** (*BaseModel*) – The BaseModel instance.
- **context** (*Any*) – The context.

Return type

None

remove_route(*route_name*)

Remove a route and all references from the semantic router.

Parameters

route_name (*str*) – Name of the route to remove.

Return type

None

route_many(*statement=None, vector=None, max_k=None, distance_threshold=None, aggregation_method=None*)

Query the semantic router with a given statement or vector for multiple matches.

Parameters

- **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

The matching routes and their details.

Return typeList[[RouteMatch](#)]**to_dict**()

Convert the SemanticRouter instance to a dictionary.

Returns

The dictionary representation of the SemanticRouter.

Return type

Dict[str, Any]

```
from redisvl.extensions.router import SemanticRouter
router = SemanticRouter(name="example_router", routes=[], redis_url="redis://
↪localhost:6379")
router_dict = router.to_dict()
```

to_yaml(*file_path, overwrite=True*)

Write the semantic router to a YAML file.

Parameters

- **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.

Return type

None

```
from redisvl.extensions.router import SemanticRouter
router = SemanticRouter(
    name="example_router",
    routes=[],
    redis_url="redis://localhost:6379"
)
router.to_yaml("router.yaml")
```

update_route_thresholds(*route_thresholds*)

Update the distance thresholds for each route.

Parameters

route_thresholds (*Dict[str, float]*) – Dictionary of route names and their distance thresholds.

update_routing_config(*routing_config*)

Update the routing configuration.

Parameters

routing_config (*RoutingConfig*) – The new routing configuration.

model_config: *ClassVar[ConfigDict]* = {'arbitrary_types_allowed': True}

Configuration for the model, should be a dictionary conforming to *[ConfigDict][pydantic.config.ConfigDict]*.

name: *str*

The name of the semantic router.

property route_names: *List[str]*

Get the list of route names.

Returns

List of route names.

Return type

List[str]

property route_thresholds: *Dict[str, float | None]*

Get the distance thresholds for each route.

Returns

Dictionary of route names and their distance thresholds.

Return type

Dict[str, float]

routes: *List[Route]*

List of Route objects.

routing_config: *RoutingConfig*

Configuration for routing behavior.

vectorizer: *BaseVectorizer*

The vectorizer used to embed route references.

Routing Config

class RoutingConfig(*, max_k=1, aggregation_method=DistanceAggregationMethod.avg)

Configuration for routing behavior.

Create a new model by parsing and validating input data from keyword arguments.

Raises [ValidationError][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **max_k** (Annotated[int, FieldInfo(annotation=NoneType, required=True, metadata=[Strict(strict=True), Gt(gt=0)])])
- **aggregation_method** (DistanceAggregationMethod)

max_k: Annotated[int, FieldInfo(annotation=NoneType, required=True, metadata=[Strict(strict=True), Gt(gt=0)])]

Aggregation method to use to classify queries.

model_config: ClassVar[ConfigDict] = {'extra': 'ignore'}

Configuration for the model, should be a dictionary conforming to [ConfigDict][pydantic.config.ConfigDict].

Route

class Route(*, name, references, metadata={}, distance_threshold=0.5)

Model representing a routing path with associated metadata and thresholds.

Create a new model by parsing and validating input data from keyword arguments.

Raises [ValidationError][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **name** (str)
- **references** (List[str])
- **metadata** (Dict[str, Any])
- **distance_threshold** (Annotated[float, FieldInfo(annotation=NoneType, required=True, metadata=[Strict(strict=True), Gt(gt=0), Le(le=2)])])

distance_threshold: Annotated[float, FieldInfo(annotation=NoneType, required=True, metadata=[Strict(strict=True), Gt(gt=0), Le(le=2)])]

Distance threshold for matching the route.

metadata: Dict[str, Any]

Metadata associated with the route.

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to [ConfigDict][pydantic.config.ConfigDict].

name: str

The name of the route.

references: List[str]

List of reference phrases for the route.

Route Match

class RouteMatch(*, name=None, distance=None)

Model representing a matched route with distance information.

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **name** (*str* | *None*)
- **distance** (*float* | *None*)

distance: float | None

The vector distance between the statement and the matched route.

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

name: str | None

The matched route name.

Distance Aggregation Method

class DistanceAggregationMethod(value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None)

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.

2.3 User Guides

User guides provide helpful resources for using RedisVL and its different components.

2.3.1 Getting Started with RedisVL

`redisvl` is a versatile Python library with an integrated CLI, designed to enhance AI applications using Redis. This guide will walk you through the following steps:

1. Defining an `IndexSchema`
2. Preparing a sample dataset
3. Creating a `SearchIndex` object
4. Testing `rvl` CLI functionality
5. Loading the sample data
6. Building `VectorQuery` objects and executing searches
7. Updating a `SearchIndex` object

...and more!

Prerequisites:

- Ensure `redisvl` is installed in your Python environment.
- Have a running instance of [Redis Stack](#) or [Redis Cloud](#).

Define an `IndexSchema`

The `IndexSchema` maintains crucial **index configuration** and **field definitions** to enable search with Redis. For ease of use, the schema can be constructed from a python dictionary or yaml file.

Example Schema Creation

Consider a dataset with user information, including `job`, `age`, `credit_score`, and a 3-dimensional `user_embedding` vector.

You must also decide on a Redis index name and key prefix to use for this dataset. Below are example schema definitions in both YAML and Dict format.

YAML Definition:

```
version: '0.1.0'

index:
  name: user_simple
  prefix: user_simple_docs

fields:
  - name: user
    type: tag
```

(continues on next page)

(continued from previous page)

```

- name: credit_score
  type: tag
- name: job
  type: text
- name: age
  type: numeric
- name: user_embedding
  type: vector
  attrs:
    algorithm: flat
    dims: 3
    distance_metric: cosine
    datatype: float32

```

Store this in a local file, such as `schema.yaml`, for RedisVL usage.

Python Dictionary:

```

schema = {
  "index": {
    "name": "user_simple",
    "prefix": "user_simple_docs",
  },
  "fields": [
    {"name": "user", "type": "tag"},
    {"name": "credit_score", "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"
      }
    }
  ]
}

```

Sample Dataset Preparation

Below, create a mock dataset with `user`, `job`, `age`, `credit_score`, and `user_embedding` fields. The `user_embedding` vectors are synthetic examples for demonstration purposes.

For more information on creating real-world embeddings, refer to this [article](#).

```
import numpy as np
```

```
data = [
```

(continues on next page)

(continued from previous page)

```

{
    'user': 'john',
    'age': 1,
    'job': 'engineer',
    'credit_score': 'high',
    'user_embedding': np.array([0.1, 0.1, 0.5], dtype=np.float32).tobytes()
},
{
    'user': 'mary',
    'age': 2,
    'job': 'doctor',
    'credit_score': 'low',
    'user_embedding': np.array([0.1, 0.1, 0.5], dtype=np.float32).tobytes()
},
{
    'user': 'joe',
    'age': 3,
    'job': 'dentist',
    'credit_score': 'medium',
    'user_embedding': np.array([0.9, 0.9, 0.1], dtype=np.float32).tobytes()
}
]

```

As seen above, the sample `user_embedding` vectors are converted into bytes. Using the NumPy, this is fairly trivial.

Create a SearchIndex

With the schema and sample dataset ready, create a `SearchIndex`.

Bring your own Redis connection instance

This is ideal in scenarios where you have custom settings on the connection instance or if your application will share a connection pool:

```

from redisvl.index import SearchIndex
from redis import Redis

client = Redis.from_url("redis://localhost:6379")
index = SearchIndex.from_dict(schema, redis_client=client, validate_on_load=True)

```

Let the index manage the connection instance

This is ideal for simple cases:

```
index = SearchIndex.from_dict(schema, redis_url="redis://localhost:6379", validate_on_
↪load=True)

# If you don't specify a client or Redis URL, the index will attempt to
# connect to Redis at the default address "redis://localhost:6379".
```

Create the index

Now that we are connected to Redis, we need to run the create command.

```
index.create(overwrite=True)
```

```
13:00:22 redisvl.index.index INFO    Index already exists, overwriting.
```

Note that at this point, the index has no entries. Data loading follows.

Inspect with the rv1 CLI

Use the rv1 CLI to inspect the created index and its fields:

```
!rv1 index listall
```

```
13:00:24 [RedisVL] INFO    Indices:
13:00:24 [RedisVL] INFO    1. user_simple
```

```
!rv1 index info -i user_simple
```

Index Information:

Index Name	Storage Type	Prefixes	Index Options
↪ Indexing			
↪ user_simple	HASH	['user_simple_docs']	[]
↪ 0			

Index Fields:

Name	Attribute	Type	Field Option	Option Value
↪ Field Option	Option Value	Field Option	Option Value	Field
↪ Option	Option Value			
↪ user	user	TAG	SEPARATOR	,
↪ credit_score	credit_score	TAG	SEPARATOR	,

(continues on next page)

(continued from previous page)

↪ job	job	TEXT	WEIGHT	1	↪
↪					↪
↪ age	age	NUMERIC			↪
↪					↪
↪ user_embedding	user_embedding	VECTOR	algorithm	FLAT	↪
↪ data_type	FLOAT32	dim	3	distance_	
↪ metric COSINE					

Load Data to SearchIndex

Load the sample dataset to Redis.

Validate data entries on load

RedisVL uses pydantic validation under the hood to ensure loaded data is valid and confirms to your schema. This setting is optional and can be configured in the `SearchIndex` class.

```
keys = index.load(data)
```

```
print(keys)
```

```
['user_simple_docs:01JY4J4Y08GFY10VMB9D4YDMZQ', 'user_simple_docs:
↪01JY4J4Y0AY2MKJ24XQS2Q2YS', 'user_simple_docs:01JY4J4Y0A9GFF2XG1R81EFD4Z']
```

By default, load will create a unique Redis key as a combination of the index key prefix and a random ULID. You can also customize the key by providing direct keys or pointing to a specified `id_field` on load.

Load INVALID data

This will raise a `SchemaValidationError` if `validate_on_load` is set to true in the `SearchIndex` class.

```
# NBVAL_SKIP
```

```
try:
    keys = index.load([{"user_embedding": True}])
except Exception as e:
    print(str(e))
```

```
13:00:27 redisvl.index.index ERROR    Data validation failed during load operation
Schema validation failed for object at index 0. Field 'user_embedding' expects bytes
↪(vector data), but got boolean value 'True'. If this should be a vector field, provide
↪a list of numbers or bytes. If this should be a different field type, check your
↪schema definition.
Object data: {
```

(continues on next page)

(continued from previous page)

```
"user_embedding": true
}
```

Hint: Check that your data types match the schema field definitions. Use `index.schema.fields` to view expected field types.

Upsert the index with new data

Upsert data by using the load method again:

```
# Add more data
new_data = [{
    'user': 'tyler',
    'age': 9,
    'job': 'engineer',
    'credit_score': 'high',
    'user_embedding': np.array([0.1, 0.3, 0.5], dtype=np.float32).tobytes()
}]
keys = index.load(new_data)

print(keys)
```

```
['user_simple_docs:01JY4J4Y0N4CNR9Y6R67MMVG7Q']
```

Creating VectorQuery Objects

Next we will create a vector query object for our newly populated index. This example will use a simple vector to demonstrate how vector similarity works. Vectors in production will likely be much larger than 3 floats and often require Machine Learning models (i.e. Huggingface sentence transformers) or an embeddings API (Cohere, OpenAI). `redisvl` provides a set of [Vectorizers](#) to assist in vector creation.

```
from redisvl.query import VectorQuery
from jupyterutils import result_print

query = VectorQuery(
    vector=[0.1, 0.1, 0.5],
    vector_field_name="user_embedding",
    return_fields=["user", "age", "job", "credit_score", "vector_distance"],
    num_results=3
)
```

Executing queries

With our VectorQuery object defined above, we can execute the query over the SearchIndex using the query method.

```
results = index.query(query)
result_print(results)
```

```
<IPython.core.display.HTML object>
```

Using an Asynchronous Redis Client

The AsyncSearchIndex class along with an async Redis python client allows for queries, index creation, and data loading to be done asynchronously. This is the recommended route for working with redisvl in production-like settings.

```
schema
```

```
{'index': {'name': 'user_simple', 'prefix': 'user_simple_docs'},
 'fields': [{'name': 'user', 'type': 'tag'},
             {'name': 'credit_score', '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'}}]}
```

```
from redisvl.index import AsyncSearchIndex
from redis.asyncio import Redis

client = Redis.from_url("redis://localhost:6379")
index = AsyncSearchIndex.from_dict(schema, redis_client=client)
```

```
# execute the vector query async
results = await index.query(query)
result_print(results)
```

```
<IPython.core.display.HTML object>
```

Updating a schema

In some scenarios, it makes sense to update the index schema. With Redis and `redisvl`, this is easy because Redis can keep the underlying data in place while you change or make updates to the index configuration.

So for our scenario, let's imagine we want to reindex this data in 2 ways:

- by using a Tag type for `job` field instead of Text
- by using an `hnsw` vector index for the `user_embedding` field instead of a flat vector index

Modify this schema to have what we want

```
index.schema.remove_field("job")
index.schema.remove_field("user_embedding")
index.schema.add_fields([
    {"name": "job", "type": "tag"},
    {
        "name": "user_embedding",
        "type": "vector",
        "attrs": {
            "dims": 3,
            "distance_metric": "cosine",
            "algorithm": "hnsw",
            "datatype": "float32"
        }
    }
])
```

Run the index update but keep underlying data in place

```
await index.create(overwrite=True, drop=False)
```

```
13:00:27 redisvl.index.index INFO Index already exists, overwriting.
```

Execute the vector query async

```
results = await index.query(query)
result_print(results)
```

```
<IPython.core.display.HTML object>
```

Check Index Stats

Use the `rvl` CLI to check the stats for the index:

```
!rvl stats -i user_simple
```

Statistics:

Stat Key	Value
num_docs	10
num_terms	0

(continues on next page)

(continued from previous page)

max_doc_id	10
num_records	50
percent_indexed	1
hash_indexing_failures	0
number_of_uses	2
bytes_per_record_avg	19.5200004
doc_table_size_mb	0.00105857
inverted_sz_mb	9.30786132
key_table_size_mb	4.70161437
offset_bits_per_record_avg	nan
offset_vectors_sz_mb	0
offsets_per_term_avg	0
records_per_doc_avg	5
sortable_values_size_mb	0
total_indexing_time	0.16899999
total_inverted_index_blocks	11
vector_index_sz_mb	0.23619842

Cleanup

Below we will clean up after our work. First, you can flush all data from Redis associated with the index by using the `.clear()` method. This will leave the secondary index in place for future insertions or updates.

But if you want to clean up everything, including the index, just use `.delete()` which will by default remove the index AND the underlying data.

```
# Clear all data from Redis associated with the index
await index.clear()
```

```
10
```

```
# Butm the index is still in place
await index.exists()
```

```
True
```

```
# Remove / delete the index in its entirety
await index.delete()
```

2.3.2 Querying with RedisVL

In this notebook, we will explore more complex queries that can be performed with `redisvl`

Before running this notebook, be sure to

1. Have installed `redisvl` and have that environment active for this notebook.
2. Have a running Redis instance with RediSearch > 2.4 running.

```
import pickle
from jupyterutils import table_print, result_print

# load in the example data and printing utils
data = pickle.load(open("hybrid_example_data.pkl", "rb"))
table_print(data)
```

<IPython.core.display.HTML object>

```
schema = {
    "index": {
        "name": "user_queries",
        "prefix": "user_queries_docs",
        "storage_type": "hash", # default setting -- HASH
    },
    "fields": [
        {"name": "user", "type": "tag"},
        {"name": "credit_score", "type": "tag"},
        {"name": "job", "type": "text"},
        {"name": "age", "type": "numeric"},
        {"name": "last_updated", "type": "numeric"},
        {"name": "office_location", "type": "geo"},
        {
            "name": "user_embedding",
            "type": "vector",
            "attrs": {
                "dims": 3,
                "distance_metric": "cosine",
                "algorithm": "flat",
                "datatype": "float32"
            }
        }
    ]
}
```

```
from redisvl.index import SearchIndex

# construct a search index from the schema
index = SearchIndex.from_dict(schema, redis_url="redis://localhost:6379")

# create the index (no data yet)
index.create(overwrite=True)
```

```
# use the CLI to see the created index
!rvl index listall
```

```
13:00:56 [RedisVL] INFO    Indices:
13:00:56 [RedisVL] INFO    1. user_queries
```

```
# load data to redis
keys = index.load(data)
```

```
index.info()['num_docs']
```

```
7
```

Hybrid Queries

Hybrid queries are queries that combine multiple types of filters. For example, you may want to search for a user that is a certain age, has a certain job, and is within a certain distance of a location. This is a hybrid query that combines numeric, tag, and geographic filters.

Tag Filters

Tag filters are filters that are applied to tag fields. These are fields that are not tokenized and are used to store a single categorical value.

```
from redisvl.query import VectorQuery
from redisvl.query.filter import Tag

t = Tag("credit_score") == "high"

v = VectorQuery(
    vector=[0.1, 0.1, 0.5],
    vector_field_name="user_embedding",
    return_fields=["user", "credit_score", "age", "job", "office_location", "last_updated"],
    filter_expression=t
)

results = index.query(v)
result_print(results)
```

```
<IPython.core.display.HTML object>
```

```
# negation
t = Tag("credit_score") != "high"

v.set_filter(t)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# use multiple tags as a list
t = Tag("credit_score") == ["high", "medium"]

v.set_filter(t)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# use multiple tags as a set (to enforce uniqueness)
t = Tag("credit_score") == set(["high", "high", "medium"])

v.set_filter(t)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

What about scenarios where you might want to dynamically generate a list of tags? Have no fear. RedisVL allows you to do this gracefully without having to check for the **empty case**. The **empty case** is when you attempt to run a Tag filter on a field with no defined values to match:

```
Tag("credit_score") == []
```

An empty filter like the one above will yield a * Redis query filter which implies the base case – there is no filter here to use.

```
# gracefully fallback to "*" filter if empty case
empty_case = Tag("credit_score") == []

v.set_filter(empty_case)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

Numeric Filters

Numeric filters are filters that are applied to numeric fields and can be used to isolate a range of values for a given field.

```
from redisvl.query.filter import Num

numeric_filter = Num("age").between(15, 35)

v.set_filter(numeric_filter)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# exact match query
numeric_filter = Num("age") == 14

v.set_filter(numeric_filter)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# negation
numeric_filter = Num("age") != 14
```

(continues on next page)

(continued from previous page)

```
v.set_filter(numeric_filter)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

Timestamp Filters

In redis all times are stored as an epoch time numeric however, this class allows you to filter with python datetime for ease of use.

```
from redisvl.query.filter import Timestamp
from datetime import datetime

dt = datetime(2025, 3, 16, 13, 45, 39, 132589)
print(f'Epoch comparison: {dt.timestamp()}')

timestamp_filter = Timestamp("last_updated") > dt

v.set_filter(timestamp_filter)
result_print(index.query(v))
```

```
Epoch comparison: 1742147139.132589
```

```
<IPython.core.display.HTML object>
```

```
from redisvl.query.filter import Timestamp
from datetime import datetime

dt = datetime(2025, 3, 16, 13, 45, 39, 132589)

print(f'Epoch comparison: {dt.timestamp()}')

timestamp_filter = Timestamp("last_updated") < dt

v.set_filter(timestamp_filter)
result_print(index.query(v))
```

```
Epoch comparison: 1742147139.132589
```

```
<IPython.core.display.HTML object>
```

```
from redisvl.query.filter import Timestamp
from datetime import datetime

dt_1 = datetime(2025, 1, 14, 13, 45, 39, 132589)
dt_2 = datetime(2025, 3, 16, 13, 45, 39, 132589)

print(f'Epoch between: {dt_1.timestamp()} - {dt_2.timestamp()}')
```

(continues on next page)

(continued from previous page)

```
timestamp_filter = Timestamp("last_updated").between(dt_1, dt_2)

v.set_filter(timestamp_filter)
result_print(index.query(v))
```

```
Epoch between: 1736880339.132589 - 1742147139.132589
```

```
<IPython.core.display.HTML object>
```

Text Filters

Text filters are filters that are applied to text fields. These filters are applied to the entire text field. For example, if you have a text field that contains the text “The quick brown fox jumps over the lazy dog”, a text filter of “quick” will match this text field.

```
from redisvl.query.filter import Text

# exact match filter -- document must contain the exact word doctor
text_filter = Text("job") == "doctor"

v.set_filter(text_filter)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# negation -- document must not contain the exact word doctor
negate_text_filter = Text("job") != "doctor"

v.set_filter(negate_text_filter)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# wildcard match filter
wildcard_filter = Text("job") % "doct*"

v.set_filter(wildcard_filter)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# fuzzy match filter
fuzzy_match = Text("job") % "%%engine%"

v.set_filter(fuzzy_match)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# conditional -- match documents with job field containing engineer OR doctor
conditional = Text("job") % "engineer|doctor"

v.set_filter(conditional)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# gracefully fallback to "*" filter if empty case
empty_case = Text("job") % ""

v.set_filter(empty_case)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

Use raw query strings as input. Below we use the ~ flag to indicate that the full text query is optional. We also choose the BM25 scorer and return document scores along with the result.

```
v.set_filter("(~(@job:engineer))")
v.scorer("BM25").with_scores()

index.query(v)
```

```
[{'id': 'user_queries_docs:01JY4J5VC91SV4C91BM4D0FCV2',
  'score': 0.9090908893868948,
  'vector_distance': '0',
  'user': 'john',
  'credit_score': 'high',
  'age': '18',
  'job': 'engineer',
  'office_location': '-122.4194,37.7749',
  'last_updated': '1741627789'},
 {'id': 'user_queries_docs:01JY4J5VC90DRSFJ0WKXXN49JT',
  'score': 0.0,
  'vector_distance': '0',
  'user': 'derrick',
  'credit_score': 'low',
  'age': '14',
  'job': 'doctor',
  'office_location': '-122.4194,37.7749',
  'last_updated': '1741627789'},
 {'id': 'user_queries_docs:01JY4J5VC9QTPMCD60YP40Q6PW',
  'score': 0.9090908893868948,
  'vector_distance': '0.109129190445',
  'user': 'tyler',
  'credit_score': 'high',
  'age': '100',
  'job': 'engineer',
```

(continues on next page)

(continued from previous page)

```

    'office_location': '-122.0839,37.3861',
    'last_updated': '1742232589'},
{'id': 'user_queries_docs:01JY4J5VC9FW7QQNJKDJ4Z7PRG',
 'score': 0.0,
 'vector_distance': '0.158808946609',
 'user': 'tim',
 'credit_score': 'high',
 'age': '12',
 'job': 'dermatologist',
 'office_location': '-122.0839,37.3861',
 'last_updated': '1739644189'},
{'id': 'user_queries_docs:01JY4J5VC940DJ9F47EJ6KN2MH',
 'score': 0.0,
 'vector_distance': '0.217882037163',
 'user': 'taimur',
 'credit_score': 'low',
 'age': '15',
 'job': 'CEO',
 'office_location': '-122.0839,37.3861',
 'last_updated': '1742232589'},
{'id': 'user_queries_docs:01JY4J5VC9D53KQD7ZTRP14KCE',
 'score': 0.0,
 'vector_distance': '0.266666650772',
 'user': 'nancy',
 'credit_score': 'high',
 'age': '94',
 'job': 'doctor',
 'office_location': '-122.4194,37.7749',
 'last_updated': '1710696589'},
{'id': 'user_queries_docs:01JY4J5VC9806MD90GBZNP0MNY',
 'score': 0.0,
 'vector_distance': '0.653301358223',
 'user': 'joe',
 'credit_score': 'medium',
 'age': '35',
 'job': 'dentist',
 'office_location': '-122.0839,37.3861',
 'last_updated': '1742232589'}]}

```

Geographic Filters

Geographic filters are filters that are applied to geographic fields. These filters are used to find results that are within a certain distance of a given point. The distance is specified in kilometers, miles, meters, or feet. A radius can also be specified to find results within a certain radius of a given point.

```

from redisvl.query.filter import Geo, GeoRadius

# within 10 km of San Francisco office
geo_filter = Geo("office_location") == GeoRadius(-122.4194, 37.7749, 10, "km")

v.set_filter(geo_filter)

```

(continues on next page)

(continued from previous page)

```
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# within 100 km Radius of San Francisco office
geo_filter = Geo("office_location") == GeoRadius(-122.4194, 37.7749, 100, "km")

v.set_filter(geo_filter)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# not within 10 km Radius of San Francisco office
geo_filter = Geo("office_location") != GeoRadius(-122.4194, 37.7749, 10, "km")

v.set_filter(geo_filter)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

Combining Filters

In this example, we will combine a numeric filter with a tag filter. We will search for users that are between the ages of 20 and 30 and have a job of “engineer”.

Intersection (“and”)

```
t = Tag("credit_score") == "high"
low = Num("age") >= 18
high = Num("age") <= 100
ts = Timestamp("last_updated") > datetime(2025, 3, 16, 13, 45, 39, 132589)

combined = t & low & high & ts

v = VectorQuery([0.1, 0.1, 0.5],
                "user_embedding",
                return_fields=["user", "credit_score", "age", "job", "office_location"],
                filter_expression=combined)

result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

Union (“or”)

The union of two queries is the set of all results that are returned by either of the two queries. The union of two queries is performed using the `|` operator.

```
low = Num("age") < 18
high = Num("age") > 93

combined = low | high

v.set_filter(combined)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

Dynamic Combination

There are often situations where you may or may not want to use a filter in a given query. As shown above, filters will except the `None` type and revert to a wildcard filter essentially returning all results.

The same goes for filter combinations which enables rapid reuse of filters in requests with different parameters as shown below. This removes the need for a number of “if-then” conditionals to test for the empty case.

```
def make_filter(age=None, credit=None, job=None):
    flexible_filter = (
        (Num("age") > age) &
        (Tag("credit_score") == credit) &
        (Text("job") % job)
    )
    return flexible_filter
```

```
# all parameters
combined = make_filter(age=18, credit="high", job="engineer")
v.set_filter(combined)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# just age and credit_score
combined = make_filter(age=18, credit="high")
v.set_filter(combined)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# just age
combined = make_filter(age=18)
v.set_filter(combined)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# no filters
combined = make_filter()
v.set_filter(combined)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

Non-vector Queries

In some cases, you may not want to run a vector query, but just use a `FilterExpression` similar to a SQL query. The `FilterQuery` class enable this functionality. It is similar to the `VectorQuery` class but solely takes a `FilterExpression`.

```
from redisvl.query import FilterQuery

has_low_credit = Tag("credit_score") == "low"

filter_query = FilterQuery(
    return_fields=["user", "credit_score", "age", "job", "location"],
    filter_expression=has_low_credit
)

results = index.query(filter_query)

result_print(results)
```

```
<IPython.core.display.HTML object>
```

Count Queries

In some cases, you may need to use a `FilterExpression` to execute a `CountQuery` that simply returns the count of the number of entities in the pertaining set. It is similar to the `FilterQuery` class but does not return the values of the underlying data.

```
from redisvl.query import CountQuery

has_low_credit = Tag("credit_score") == "low"

filter_query = CountQuery(filter_expression=has_low_credit)

count = index.query(filter_query)

print(f"{count} records match the filter expression {str(has_low_credit)} for the given_
↪index.")
```

```
2 records match the filter expression @credit_score:{low} for the given index.
```

Range Queries

Range Queries are a useful method to perform a vector search where only results within a vector `distance_threshold` are returned. This enables the user to find all records within their dataset that are similar to a query vector where “similar” is defined by a quantitative value.

```
from redisvl.query import RangeQuery

range_query = 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
)

# same as the vector query or filter query
results = index.query(range_query)

result_print(results)
```

```
<IPython.core.display.HTML object>
```

We can also change the distance threshold of the query object between uses if we like. Here we will set `distance_threshold==0.1`. This means that the query object will return all matches that are within 0.1 of the query object. This is a small distance, so we expect to get fewer matches than before.

```
range_query.set_distance_threshold(0.1)

result_print(index.query(range_query))
```

```
<IPython.core.display.HTML object>
```

Range queries can also be used with filters like any other query type. The following limits the results to only include records with a job of engineer while also being within the vector range (aka distance).

```
is_engineer = Text("job") == "engineer"

range_query.set_filter(is_engineer)

result_print(index.query(range_query))
```

```
<IPython.core.display.HTML object>
```

Advanced Query Modifiers

See all modifier options available on the query API docs: <https://docs.redisvl.com/en/latest/api/query.html>

```
# Sort by a different field and change dialect
v = VectorQuery(
    vector=[0.1, 0.1, 0.5],
    vector_field_name="user_embedding",
    return_fields=["user", "credit_score", "age", "job", "office_location"],
    num_results=5,
    filter_expression=is_engineer
).sort_by("age", asc=False).dialect(3)

result = index.query(v)
result_print(result)
```

```
<IPython.core.display.HTML object>
```

Raw Redis Query String

Sometimes it's helpful to convert these classes into their raw Redis query strings.

```
# check out the complex query from above
str(v)
```

```
'@job:("engineer")=>[KNN 5 @user_embedding $vector AS vector_distance] RETURN 6 user_
↪credit_score age job office_location vector_distance SORTBY age DESC DIALECT 3 LIMIT 0_
↪5'
```

```
t = Tag("credit_score") == "high"

str(t)
```

```
'@credit_score:{high}'
```

```
t = Tag("credit_score") == "high"
low = Num("age") >= 18
high = Num("age") <= 100

combined = t & low & high

str(combined)
```

```
'((@credit_score:{high} @age:[18 +inf]) @age:[-inf 100])'
```

The RedisVL SearchIndex class exposes a search() method which is a simple wrapper around the FT.SEARCH API. Provide any valid Redis query string.

```
results = index.search(str(t))
for r in results.docs:
    print(r.__dict__)
```

```
{'id': 'user_queries_docs:01JY4J5VC91SV4C91BM4D0FCV2', 'payload': None, 'user': 'john',
→ 'age': '18', 'job': 'engineer', 'credit_score': 'high', 'office_location': '-122.4194,
→ 37.7749', 'user_embedding': '==\x00\x00\x00?', 'last_updated': '1741627789'}
{'id': 'user_queries_docs:01JY4J5VC9D53KQD7ZTRP14KCE', 'payload': None, 'user': 'nancy',
→ 'age': '94', 'job': 'doctor', 'credit_score': 'high', 'office_location': '-122.4194,37.
→ 7749', 'user_embedding': '333?=\x00\x00\x00?', 'last_updated': '1710696589'}
{'id': 'user_queries_docs:01JY4J5VC9QTPMCD60YP40Q6PW', 'payload': None, 'user': 'tyler',
→ 'age': '100', 'job': 'engineer', 'credit_score': 'high', 'office_location': '-122.0839,
→ 37.3861', 'user_embedding': '=>\x00\x00\x00?', 'last_updated': '1742232589'}
{'id': 'user_queries_docs:01JY4J5VC9FW7QQNJKD4Z7PRG', 'payload': None, 'user': 'tim',
→ 'age': '12', 'job': 'dermatologist', 'credit_score': 'high', 'office_location': '-122.
→ 0839,37.3861', 'user_embedding': '>>\x00\x00\x00?', 'last_updated': '1739644189'}
```

```
# Cleanup
index.delete()
```

2.3.3 LLM Caching

This notebook demonstrates how to use RedisVL's `SemanticCache` to cache LLM responses based on semantic similarity. Semantic caching can significantly reduce API costs and latency by retrieving cached responses for semantically similar prompts instead of making redundant API calls.

Key features covered:

- Basic cache operations (store, check, clear)
- Customizing semantic similarity thresholds
- TTL policies for cache expiration
- Performance benchmarking
- Access controls with tags and filters for multi-user scenarios

Prerequisites:

- Ensure `redisvl` is installed in your Python environment
- Have a running instance of [Redis Stack](#) or [Redis Cloud](#)
- OpenAI API key for the examples

First, we will import `OpenAI` to use their API for responding to user prompts. We will also create a simple `ask_openai` helper method to assist.

```
import os
import getpass
import time
import numpy as np

from openai import OpenAI

os.environ["TOKENIZERS_PARALLELISM"] = "False"

api_key = os.getenv("OPENAI_API_KEY") or getpass.getpass("Enter your OpenAI API key: ")
```

(continues on next page)

(continued from previous page)

```

client = OpenAI(api_key=api_key)

def ask_openai(question: str) -> str:
    response = client.completions.create(
        model="gpt-3.5-turbo-instruct",
        prompt=question,
        max_tokens=200
    )
    return response.choices[0].text.strip()

```

```

# Test
print(ask_openai("What is the capital of France?"))

```

The capital of France is Paris.

Initializing SemanticCache

SemanticCache will automatically create an index within Redis upon initialization for the semantic cache content.

```

import warnings
warnings.filterwarnings('ignore')

from redisvl.extensions.cache.llm import SemanticCache
from redisvl.utils.vectorize import HFTextVectorizer

llmcache = SemanticCache(
    name="llmcache", # underlying search index
    redis_url="redis://localhost:6379", # redis connection url
    distance_threshold=0.1, # semantic cache distance
    vectorizer=HFTextVectorizer("redis/langcache-embed-v1"), # embedding model
)

```

```

13:02:02 sentence_transformers.SentenceTransformer INFO    Use pytorch device_name: mps
13:02:02 sentence_transformers.SentenceTransformer INFO    Load pretrained
→SentenceTransformer: redis/langcache-embed-v1
13:02:02 sentence_transformers.SentenceTransformer WARNING  You try to use a model that
→was created with version 4.1.0, however, your version is 3.4.1. This might cause
→unexpected behavior or errors. In that case, try to update to the latest version.

```

```
Batches: 100%| 1/1 [00:00<00:00, 3.79it/s]
```

```

# look at the index specification created for the semantic cache lookup
!rvm index info -i llmcache

```

Index Information:

(continues on next page)

(continued from previous page)

Index Name	Storage Type	Prefixes	Index Options	Indexing	
llmcache	HASH	['llmcache']	[]	0	
Index Fields:					
Name	Attribute	Type	Field Option	Option Value	
Field Option	Option Value	Field Option	Option Value	Field	
Option	Option Value				
prompt	prompt	TEXT	WEIGHT	1	
response	response	TEXT	WEIGHT	1	
inserted_at	inserted_at	NUMERIC			
updated_at	updated_at	NUMERIC			
prompt_vector	prompt_vector	VECTOR	algorithm	FLAT	
data_type	FLOAT32	dim	768	distance_	
metric	COSINE				

Basic Cache Usage

```
question = "What is the capital of France?"

# Check the semantic cache -- should be empty
if response := llmcache.check(prompt=question):
    print(response)
else:
    print("Empty cache")

Batches: 100%| 1/1 [00:00<00:00, 7.79it/s]

Empty cache
```

Our initial cache check should be empty since we have not yet stored anything in the cache. Below, store the question, proper response, and any arbitrary metadata (as a python dictionary object) in the cache.

```
# Cache the question, answer, and arbitrary metadata
llmcache.store(
    prompt=question,
```

(continues on next page)

(continued from previous page)

```
response="Paris",
metadata={"city": "Paris", "country": "france"}
)
```

```
Batches: 100%| 1/1 [00:00<00:00, 19.62it/s]
```

```
'llmcache:115049a298532be2f181edb03f766770c0db84c22aff39003fec340deaec7545'
```

Now we will check the cache again with the same question and with a semantically similar question:

```
# Check the cache again
if response := llmcache.check(prompt=question, return_fields=["prompt", "response",
↪ "metadata"]):
    print(response)
else:
    print("Empty cache")
```

```
Batches: 100%| 1/1 [00:00<00:00, 18.65it/s]
```

```
[{'prompt': 'What is the capital of France?', 'response': 'Paris', 'metadata': {'city':
↪ 'Paris', 'country': 'france'}, 'key': 'llmcache:
↪ 115049a298532be2f181edb03f766770c0db84c22aff39003fec340deaec7545'}]
```

```
# Check for a semantically similar result
question = "What actually is the capital of France?"
llmcache.check(prompt=question)[0]['response']
```

```
Batches: 100%| 1/1 [00:00<00:00, 7.81it/s]
```

```
'Paris'
```

Customize the Distance Threshold

For most use cases, the right semantic similarity threshold is not a fixed quantity. Depending on the choice of embedding model, the properties of the input query, and even business use case – the threshold might need to change.

Fortunately, you can seamlessly adjust the threshold at any point like below:

```
# Widen the semantic distance threshold
llmcache.set_threshold(0.5)
```

```
# Really try to trick it by asking around the point
# But is able to slip just under our new threshold
question = "What is the capital city of the country in Europe that also has a city named_
↪ Nice?"
llmcache.check(prompt=question)[0]['response']
```

```
Batches: 100%| 1/1 [00:00<00:00, 8.37it/s]
```

```
'Paris'
```

```
# Invalidate the cache completely by clearing it out
llmcache.clear()

# Should be empty now
llmcache.check(prompt=question)
```

```
Batches: 100%| 1/1 [00:00<00:00, 21.23it/s]
```

```
[]
```

Utilize TTL

Redis uses TTL policies (optional) to expire individual keys at points in time in the future. This allows you to focus on your data flow and business logic without bothering with complex cleanup tasks.

A TTL policy set on the SemanticCache allows you to temporarily hold onto cache entries. Below, we will set the TTL policy to 5 seconds.

```
llmcache.set_ttl(5) # 5 seconds
```

```
llmcache.store("This is a TTL test", "This is a TTL test response")

time.sleep(6)
```

```
Batches: 100%| 1/1 [00:00<00:00, 8.53it/s]
```

```
# confirm that the cache has cleared by now on it's own
result = llmcache.check("This is a TTL test")

print(result)
```

```
Batches: 100%| 1/1 [00:00<00:00, 12.54it/s]
```

```
[]
```

```
# Reset the TTL to null (long lived data)
llmcache.set_ttl()
```

Simple Performance Testing

Next, we will measure the speedup obtained by using SemanticCache. We will use the `time` module to measure the time taken to generate responses with and without SemanticCache.

```
def answer_question(question: str) -> str:
    """Helper function to answer a simple question using OpenAI with a wrapper
    check for the answer in the semantic cache first.

    Args:
        question (str): User input question.

    Returns:
        str: Response.
    """
    results = llmcache.check(prompt=question)
    if results:
        return results[0]["response"]
    else:
        answer = ask_openai(question)
        return answer

start = time.time()
# asking a question -- openai response time
question = "What was the name of the first US President?"
answer = answer_question(question)
end = time.time()

print(f"Without caching, a call to openAI to answer this simple question took {end-start}
↪ seconds.")

# add the entry to our LLM cache
llmcache.store(prompt=question, response="George Washington")

Batches: 100%|| 1/1 [00:00<00:00, 8.09it/s]

13:02:17 httpx INFO HTTP Request: POST https://api.openai.com/v1/completions "HTTP/1.1
↪ 200 OK"
Without caching, a call to openAI to answer this simple question took 1.7948627471923828
↪ seconds.

Batches: 100%|| 1/1 [00:00<00:00, 12.93it/s]

'llmcache:67e0f6e28fe2a61c0022fd42bf734bb8ffe49d3e375fd69d692574295a20fc1a'

# Calculate the avg latency for caching over LLM usage
times = []

for _ in range(10):
    cached_start = time.time()
    cached_answer = answer_question(question)
    cached_end = time.time()
```

(continues on next page)

(continued from previous page)

```

times.append(cached_end-cached_start)

avg_time_with_cache = np.mean(times)
print(f"Avg time taken with LLM cache enabled: {avg_time_with_cache}")
print(f"Percentage of time saved: {round(((end - start) - avg_time_with_cache) / (end -
→start) * 100, 2)}%")

```

```

Batches: 100%| 1/1 [00:00<00:00, 20.90it/s]
Batches: 100%| 1/1 [00:00<00:00, 23.24it/s]
Batches: 100%| 1/1 [00:00<00:00, 22.85it/s]
Batches: 100%| 1/1 [00:00<00:00, 21.98it/s]
Batches: 100%| 1/1 [00:00<00:00, 22.65it/s]
Batches: 100%| 1/1 [00:00<00:00, 22.65it/s]
Batches: 100%| 1/1 [00:00<00:00, 21.84it/s]
Batches: 100%| 1/1 [00:00<00:00, 20.67it/s]
Batches: 100%| 1/1 [00:00<00:00, 22.08it/s]
Batches: 100%| 1/1 [00:00<00:00, 21.14it/s]

```

```

Avg time taken with LLM cache enabled: 0.049193501472473145
Percentage of time saved: 97.26%

```

```

# check the stats of the index
!rvl stats -i llmcache

```

Statistics:

Stat Key	Value
num_docs	1
num_terms	19
max_doc_id	3
num_records	29
percent_indexed	1
hash_indexing_failures	0
number_of_uses	19
bytes_per_record_avg	75.9655151
doc_table_size_mb	1.34468078
inverted_sz_mb	0.00210094
key_table_size_mb	2.76565551
offset_bits_per_record_avg	8
offset_vectors_sz_mb	2.09808349
offsets_per_term_avg	0.75862067
records_per_doc_avg	29
sortable_values_size_mb	0
total_indexing_time	14.3260002
total_inverted_index_blocks	21
vector_index_sz_mb	3.01609802

```

# Clear the cache AND delete the underlying index
llmcache.delete()

```

Cache Access Controls, Tags & Filters

When running complex workflows with similar applications, or handling multiple users it's important to keep data segregated. Building on top of RedisVL's support for complex and hybrid queries we can tag and filter cache entries using custom-defined `filterable_fields`.

Let's store multiple users' data in our cache with similar prompts and ensure we return only the correct user information:

```
private_cache = SemanticCache(
    name="private_cache",
    filterable_fields=[{"name": "user_id", "type": "tag"}]
)

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": "abc"},
)

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": "def"},
)
```

```
13:02:20 sentence_transformers.SentenceTransformer INFO    Use pytorch device_name: mps
13:02:20 sentence_transformers.SentenceTransformer INFO    Load pretrained_
↳SentenceTransformer: redis/langcache-embed-v1
13:02:20 sentence_transformers.SentenceTransformer WARNING  You try to use a model that_
↳was created with version 4.1.0, however, your version is 3.4.1. This might cause_
↳unexpected behavior or errors. In that case, try to update to the latest version.
```

```
Batches: 100%| 1/1 [00:00<00:00, 17.15it/s]
Batches: 100%| 1/1 [00:00<00:00, 21.23it/s]
Batches: 100%| 1/1 [00:00<00:00, 21.71it/s]
```

```
'private_cache:2831a0659fb888e203cd9fedb9f65681bfa55e4977c092ed1bf87d42d2655081'
```

```
from redisvl.query.filter import Tag

# define user id filter
user_id_filter = Tag("user_id") == "abc"

response = private_cache.check(
    prompt="What is the phone number linked to my account?",
    filter_expression=user_id_filter,
    num_results=2
)

print(f"found {len(response)} entry \n{response[0]['response']}")
```

```
Batches: 100%| 1/1 [00:00<00:00, 22.36it/s]
```

```
found 1 entry
The number on file is 123-555-0000
```

```
# Cleanup
private_cache.delete()
```

Multiple filterable_fields can be defined on a cache, and complex filter expressions can be constructed to filter on these fields, as well as the default fields already present.

```
complex_cache = SemanticCache(
    name='account_data',
    filterable_fields=[
        {"name": "user_id", "type": "tag"},
        {"name": "account_type", "type": "tag"},
        {"name": "account_balance", "type": "numeric"},
        {"name": "transaction_amount", "type": "numeric"}
    ]
)
complex_cache.store(
    prompt="what is my most recent checking account transaction under $100?",
    response="Your most recent transaction was for $75",
    filters={"user_id": "abc", "account_type": "checking", "transaction_amount": 75},
)
complex_cache.store(
    prompt="what is my most recent savings account transaction?",
    response="Your most recent deposit was for $300",
    filters={"user_id": "abc", "account_type": "savings", "transaction_amount": 300},
)
complex_cache.store(
    prompt="what is my most recent checking account transaction over $200?",
    response="Your most recent transaction was for $350",
    filters={"user_id": "abc", "account_type": "checking", "transaction_amount": 350},
)
complex_cache.store(
    prompt="what is my checking account balance?",
    response="Your current checking account is $1850",
    filters={"user_id": "abc", "account_type": "checking"},
)
```

```
13:02:21 sentence_transformers.SentenceTransformer INFO    Use pytorch device_name: mps
13:02:21 sentence_transformers.SentenceTransformer INFO    Load pretrained_
↳SentenceTransformer: redis/langcache-embed-v1
13:02:21 sentence_transformers.SentenceTransformer WARNING You try to use a model that_
↳was created with version 4.1.0, however, your version is 3.4.1. This might cause_
↳unexpected behavior or errors. In that case, try to update to the latest version.
```

```
Batches: 100%| 1/1 [00:00<00:00, 21.08it/s]
Batches: 100%| 1/1 [00:00<00:00, 8.74it/s]
Batches: 100%| 1/1 [00:00<00:00, 8.01it/s]
Batches: 100%| 1/1 [00:00<00:00, 21.70it/s]
```

(continues on next page)

(continued from previous page)

```
Batches: 100%|| 1/1 [00:00<00:00, 16.74it/s]
```

```
'account_data:944f89729b09ca46b99923d223db45e0bccf584cf53fcaf87d2a58f072582d3'
```

```
from redisvl.query.filter import Num

value_filter = Num("transaction_amount") > 100
account_filter = Tag("account_type") == "checking"
complex_filter = value_filter & account_filter

# check for checking account transactions over $100
complex_cache.set_threshold(0.3)
response = complex_cache.check(
    prompt="what is my most recent checking account transaction?",
    filter_expression=complex_filter,
    num_results=5
)
print(f'found {len(response)} entry')
print(response[0]["response"])
```

```
Batches: 100%|| 1/1 [00:00<00:00, 19.91it/s]
```

```
found 1 entry
Your most recent transaction was for $350
```

```
# Cleanup
complex_cache.delete()
```

2.3.4 Vectorizers

In this notebook, we will show how to use RedisVL to create embeddings using the built-in text embedding vectorizers. Today RedisVL supports:

1. OpenAI
2. HuggingFace
3. Vertex AI
4. Cohere
5. Mistral AI
6. Amazon Bedrock
7. Bringing your own vectorizer
8. VoyageAI

Before running this notebook, be sure to

1. Have installed `redisvl` and have that environment active for this notebook.

2. Have a running Redis Stack instance with RedisSearch > 2.4 active.

For example, you can run Redis Stack locally with Docker:

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

This will run Redis on port 6379 and RedisInsight at <http://localhost:8001>.

```
# import necessary modules
import os
```

Creating Text Embeddings

This example will show how to create an embedding from 3 simple sentences with a number of different text vectorizers in RedisVL.

- “That is a happy dog”
- “That is a happy person”
- “Today is a nice day”

OpenAI

The `OpenAITextVectorizer` makes it simple to use RedisVL with the embeddings models at OpenAI. For this you will need to install `openai`.

```
pip install openai
```

```
import getpass

# setup the API Key
api_key = os.environ.get("OPENAI_API_KEY") or getpass.getpass("Enter your OpenAI API key:
→ ")
```

```
from redisvl.utils.vectorize import OpenAITextVectorizer

# create a vectorizer
oai = OpenAITextVectorizer(
    model="text-embedding-ada-002",
    api_config={"api_key": api_key},
)

test = oai.embed("This is a test sentence.")
print("Vector dimensions: ", len(test))
test[:10]
```

```
Vector dimensions: 1536
```

```
[-0.0011391325388103724,
 -0.003206387162208557,
 0.002380132209509611,
```

(continues on next page)

(continued from previous page)

```
-0.004501554183661938,
-0.010328996926546097,
0.012922565452754498,
-0.005491119809448719,
-0.0029864837415516376,
-0.007327961269766092,
-0.03365817293524742]
```

```
# Create many embeddings at once
sentences = [
    "That is a happy dog",
    "That is a happy person",
    "Today is a sunny day"
]

embeddings = oai.embed_many(sentences)
embeddings[0][:10]
```

```
[-0.017466850578784943,
 1.8471690054866485e-05,
 0.00129731057677418,
 -0.02555876597762108,
 -0.019842341542243958,
 0.01603139191865921,
 -0.0037347301840782166,
 0.0009670283179730177,
 0.006618348415941,
 -0.02497442066669464]
```

```
# openai also supports asynchronous requests, which we can use to speed up the
↳ vectorization process.
embeddings = await oai.aembed_many(sentences)
print("Number of Embeddings:", len(embeddings))
```

```
Number of Embeddings: 3
```

Azure OpenAI

The `AzureOpenAITextVectorizer` is a variation of the OpenAI vectorizer that calls OpenAI models within Azure. If you've already installed `openai`, then you're ready to use Azure OpenAI.

The only practical difference between OpenAI and Azure OpenAI is the variables required to call the API.

```
# additionally to the API Key, setup the API endpoint and version
api_key = os.environ.get("AZURE_OPENAI_API_KEY") or getpass.getpass("Enter your
↳ AzureOpenAI API key: ")
api_version = os.environ.get("OPENAI_API_VERSION") or getpass.getpass("Enter your
↳ AzureOpenAI API version: ")
azure_endpoint = os.environ.get("AZURE_OPENAI_ENDPOINT") or getpass.getpass("Enter your
↳ AzureOpenAI API endpoint: ")
```

(continues on next page)

(continued from previous page)

```
deployment_name = os.environ.get("AZURE_OPENAI_DEPLOYMENT_NAME", "text-embedding-ada-002")
↪")
```

```
from redisvl.utils.vectorize import AzureOpenAITextVectorizer

# create a vectorizer
az_oai = AzureOpenAITextVectorizer(
    model=deployment_name, # Must be your CUSTOM deployment name
    api_config={
        "api_key": api_key,
        "api_version": api_version,
        "azure_endpoint": azure_endpoint
    },
)

test = az_oai.embed("This is a test sentence.")
print("Vector dimensions: ", len(test))
test[:10]
```

```
-----
ValueError                                Traceback (most recent call last)
Cell In[7], line 4
      1 from redisvl.utils.vectorize import AzureOpenAITextVectorizer
      3 # create a vectorizer
----> 4 az_oai = AzureOpenAITextVectorizer(
      5     model=deployment_name, # Must be your CUSTOM deployment name
      6     api_config={
      7         "api_key": api_key,
      8         "api_version": api_version,
      9         "azure_endpoint": azure_endpoint
     10     },
     11 )
     13 test = az_oai.embed("This is a test sentence.")
     14 print("Vector dimensions: ", len(test))

File ~/src/redis-vl-python/redisvl/utils/vectorize/text/azureopenai.py:78, in_
↪AzureOpenAITextVectorizer.__init__(self, model, api_config, dtype)
     54 def __init__(
     55     self,
     56     model: str = "text-embedding-ada-002",
     57     api_config: Optional[Dict] = None,
     58     dtype: str = "float32",
     59 ):
     60     """Initialize the AzureOpenAI vectorizer.
     61
     62     Args:
     63     (...)
     76         ValueError: If an invalid dtype is provided.
     77     """
----> 78     self._initialize_clients(api_config)
     79     super().__init__(model=model, dims=self._set_model_dims(model), dtype=dtype)
```

(continues on next page)

(continued from previous page)

```

File ~/src/redis-vl-python/redisvl/utils/vectorize/text/azureopenai.py:106, in_
-> AzureOpenAITextVectorizer._initialize_clients(self, api_config)
    99 azure_endpoint = (
   100     api_config.pop("azure_endpoint")
   101     if api_config
   102     else os.getenv("AZURE_OPENAI_ENDPOINT")
   103 )
   105 if not azure_endpoint:
--> 106     raise ValueError(
   107         "AzureOpenAI API endpoint is required. "
   108         "Provide it in api_config or set the AZURE_OPENAI_ENDPOINT\
   109         environment variable."
   110     )
   112 api_version = (
   113     api_config.pop("api_version")
   114     if api_config
   115     else os.getenv("OPENAI_API_VERSION")
   116 )
   118 if not api_version:

ValueError: AzureOpenAI API endpoint is required. Provide it in api_config or set the_
-> AZURE_OPENAI_ENDPOINT environment variable.

```

```

# Just like OpenAI, AzureOpenAI supports batching embeddings and asynchronous requests.
sentences = [
    "That is a happy dog",
    "That is a happy person",
    "Today is a sunny day"
]

embeddings = await az_oai.aembed_many(sentences)
embeddings[0][:10]

```

Huggingface

Huggingface is a popular NLP platform that has a number of pre-trained models you can use off the shelf. RedisVL supports using Huggingface “Sentence Transformers” to create embeddings from text. To use Huggingface, you will need to install the sentence-transformers library.

```
pip install sentence-transformers
```

```

os.environ["TOKENIZERS_PARALLELISM"] = "false"
from redisvl.utils.vectorize import HFTextVectorizer

# create a vectorizer
# choose your model from the huggingface website
hf = HFTextVectorizer(model="sentence-transformers/all-mpnet-base-v2")

```

(continues on next page)

(continued from previous page)

```
# embed a sentence
test = hf.embed("This is a test sentence.")
test[:10]
```

```
# You can also create many embeddings at once
embeddings = hf.embed_many(sentences, as_buffer=True)
```

VertexAI

VertexAI is GCP's fully-featured AI platform including a number of pretrained LLMs. RedisVL supports using VertexAI to create embeddings from these models. To use VertexAI, you will first need to install the `google-cloud-aiplatform` library.

```
pip install google-cloud-aiplatform>=1.26
```

1. Then you need to gain access to a [Google Cloud Project](#) and provide [access to credentials](#). This is accomplished by setting the `GOOGLE_APPLICATION_CREDENTIALS` environment variable pointing to the path of a JSON key file downloaded from your service account on GCP.
2. Lastly, you need to find your [project ID](#) and [geographic region for VertexAI](#).

Make sure the following env vars are set:

```
GOOGLE_APPLICATION_CREDENTIALS=<path to your gcp JSON creds>
GCP_PROJECT_ID=<your gcp project id>
GCP_LOCATION=<your gcp geo region for vertex ai>
```

```
from redisvl.utils.vectorize import VertexAITextVectorizer

# create a vectorizer
vtx = VertexAITextVectorizer(api_config={
    "project_id": os.environ.get("GCP_PROJECT_ID") or getpass.getpass("Enter your GCP_
↳Project ID: "),
    "location": os.environ.get("GCP_LOCATION") or getpass.getpass("Enter your GCP_
↳Location: "),
    "google_application_credentials": os.environ.get("GOOGLE_APPLICATION_CREDENTIALS")_
↳or getpass.getpass("Enter your Google App Credentials path: ")
})

# embed a sentence
test = vtx.embed("This is a test sentence.")
test[:10]
```

Cohere

Cohere allows you to implement language AI into your product. The `CohereTextVectorizer` makes it simple to use RedisVL with the embeddings models at Cohere. For this you will need to install `cohere`.

```
pip install cohere
```

```
import getpass
# setup the API Key
api_key = os.environ.get("COHERE_API_KEY") or getpass.getpass("Enter your Cohere API key:
→ ")
```

Special attention needs to be paid to the `input_type` parameter for each `embed` call. For example, for embedding queries, you should set `input_type='search_query'`; for embedding documents, set `input_type='search_document'`. See more information [here](#)

```
from redisvl.utils.vectorize import CohereTextVectorizer

# create a vectorizer
co = CohereTextVectorizer(
    model="embed-english-v3.0",
    api_config={"api_key": api_key},
)

# embed a search query
test = co.embed("This is a test sentence.", input_type='search_query')
print("Vector dimensions: ", len(test))
print(test[:10])

# embed a document
test = co.embed("This is a test sentence.", input_type='search_document')
print("Vector dimensions: ", len(test))
print(test[:10])
```

Learn more about using RedisVL and Cohere together through [this dedicated user guide](#).

VoyageAI

VoyageAI allows you to implement language AI into your product. The `VoyageAITextVectorizer` makes it simple to use RedisVL with the embeddings models at VoyageAI. For this you will need to install `voyageai`.

```
pip install voyageai
```

```
import getpass
# setup the API Key
api_key = os.environ.get("VOYAGE_API_KEY") or getpass.getpass("Enter your VoyageAI API_
→ key: ")
```

Special attention needs to be paid to the `input_type` parameter for each `embed` call. For example, for embedding queries, you should set `input_type='query'`; for embedding documents, set `input_type='document'`. See more information [here](#)

```
from redisvl.utils.vectorize import VoyageAITextVectorizer

# create a vectorizer
vo = VoyageAITextVectorizer(
    model="voyage-law-2", # Please check the available models at https://docs.voyageai.
    ↪com/docs/embeddings
    api_config={"api_key": api_key},
)

# embed a search query
test = vo.embed("This is a test sentence.", input_type='query')
print("Vector dimensions: ", len(test))
print(test[:10])

# embed a document
test = vo.embed("This is a test sentence.", input_type='document')
print("Vector dimensions: ", len(test))
print(test[:10])
```

Mistral AI

Mistral offers LLM and embedding APIs for you to implement into your product. The `MistralAITextVectorizer` makes it simple to use RedisVL with their embeddings model. You will need to install `mistralai`.

```
pip install mistralai
```

```
from redisvl.utils.vectorize import MistralAITextVectorizer

mistral = MistralAITextVectorizer()

# embed a sentence using their asynchronous method
test = await mistral.aembed("This is a test sentence.")
print("Vector dimensions: ", len(test))
print(test[:10])
```

Amazon Bedrock

Amazon Bedrock provides fully managed foundation models for text embeddings. Install the required dependencies:

```
pip install 'redisvl[bedrock]' # Installs boto3
```

Configure AWS credentials:

```
import os
import getpass

if "AWS_ACCESS_KEY_ID" not in os.environ:
    os.environ["AWS_ACCESS_KEY_ID"] = getpass.getpass("Enter AWS Access Key ID: ")
if "AWS_SECRET_ACCESS_KEY" not in os.environ:
    os.environ["AWS_SECRET_ACCESS_KEY"] = getpass.getpass("Enter AWS Secret Key: ")

os.environ["AWS_REGION"] = "us-east-1" # Change as needed
```

Create embeddings:

```
from redisvl.utils.vectorize import BedrockTextVectorizer

bedrock = BedrockTextVectorizer(
    model="amazon.titan-embed-text-v2:0"
)

# Single embedding
text = "This is a test sentence."
embedding = bedrock.embed(text)
print(f"Vector dimensions: {len(embedding)}")

# Multiple embeddings
sentences = [
    "That is a happy dog",
    "That is a happy person",
    "Today is a sunny day"
]
embeddings = bedrock.embed_many(sentences)
```

Custom Vectorizers

RedisVL supports the use of other vectorizers and provides a class to enable compatibility with any function that generates a vector or vectors from string data

```
from redisvl.utils.vectorize import CustomTextVectorizer

def generate_embeddings(text_input, **kwargs):
    return [0.101] * 768

custom_vectorizer = CustomTextVectorizer(generate_embeddings)

custom_vectorizer.embed("This is a test sentence.")[:10]
```

This enables the use of custom vectorizers with other RedisVL components

```
from redisvl.extensions.cache.llm import SemanticCache

cache = SemanticCache(name="custom_cache", vectorizer=custom_vectorizer)

cache.store("this is a test prompt", "this is a test response")
cache.check("this is also a test prompt")
```

Search with Provider Embeddings

Now that we've created our embeddings, we can use them to search for similar sentences. We will use the same 3 sentences from above and search for similar sentences.

First, we need to create the schema for our index.

Here's what the schema for the example looks like in yaml for the HuggingFace vectorizer:

```
version: '0.1.0'

index:
  name: vectorizers
  prefix: doc
  storage_type: hash

fields:
  - name: sentence
    type: text
  - name: embedding
    type: vector
    attrs:
      dims: 768
      algorithm: flat
      distance_metric: cosine
```

```
from redisvl.index import SearchIndex

# construct a search index from the schema
index = SearchIndex.from_yaml("./schema.yaml", redis_url="redis://localhost:6379")

# create the index (no data yet)
index.create(overwrite=True)
```

```
# use the CLI to see the created index
!rvm index listall
```

Loading data to RedisVL is easy. It expects a list of dictionaries. The vector is stored as bytes.

```
from redisvl.redis.utils import array_to_buffer

embeddings = hf.embed_many(sentences)

data = [{"text": t,
        "embedding": array_to_buffer(v, dtype="float32")}]
```

(continues on next page)

(continued from previous page)

```

        for t, v in zip(sentences, embeddings)]

index.load(data)

from redisvl.query import VectorQuery

# use the HuggingFace vectorizer again to create a query embedding
query_embedding = hf.embed("That is a happy cat")

query = VectorQuery(
    vector=query_embedding,
    vector_field_name="embedding",
    return_fields=["text"],
    num_results=3
)

results = index.query(query)
for doc in results:
    print(doc["text"], doc["vector_distance"])

```

Selecting your float data type

When embedding text as byte arrays RedisVL supports 4 different floating point data types, float16, float32, float64 and bfloat16, and 2 integer types, int8 and uint8. Your dtype set for your vectorizer must match what is defined in your search index. If one is not explicitly set the default is float32.

```

vectorizer = HFTextVectorizer(dtype="float16")

# subsequent calls to embed(", as_buffer=True) and embed_many(", as_buffer=True) will now
↳ encode as float16
float16_bytes = vectorizer.embed('test sentence', as_buffer=True)

# to generate embeddings with different dtype instantiate a new vectorizer
vectorizer_64 = HFTextVectorizer(dtype='float64')
float64_bytes = vectorizer_64.embed('test sentence', as_buffer=True)

float16_bytes != float64_bytes

# cleanup
index.delete()

```

2.3.5 Hash vs JSON Storage

Out of the box, Redis provides a [variety of data structures](#) that can adapt to your domain specific applications and use cases. In this notebook, we will demonstrate how to use RedisVL with both [Hash](#) and [JSON](#) data.

Before running this notebook, be sure to

1. Have installed `redisvl` and have that environment active for this notebook.
2. Have a running Redis Stack or Redis Enterprise instance with RediSearch > 2.4 activated.

For example, you can run [Redis Stack](#) locally with Docker:

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

Or create a [FREE Redis Cloud](#).

```
# import necessary modules
import pickle

from redisvl.redis.utils import buffer_to_array
from redisvl.index import SearchIndex

# load in the example data and printing utils
data = pickle.load(open("hybrid_example_data.pkl", "rb"))
```

```
from jupyterutils import result_print, table_print

table_print(data)
```

```
<IPython.core.display.HTML object>
```

Hash or JSON – how to choose?

Both storage options offer a variety of features and tradeoffs. Below we will work through a dummy dataset to learn when and how to use both.

Working with Hashes

Hashes in Redis are simple collections of field-value pairs. Think of it like a mutable single-level dictionary contains multiple “rows”:

```
{
  "model": "Deimos",
  "brand": "Ergonom",
  "type": "Enduro bikes",
  "price": 4972,
}
```

Hashes are best suited for use cases with the following characteristics:

- Performance (speed) and storage space (memory consumption) are top concerns
- Data can be easily normalized and modeled as a single-level dict

Hashes are typically the default recommendation.

```
# define the hash index schema
hash_schema = {
    "index": {
        "name": "user-hash",
        "prefix": "user-hash-docs",
        "storage_type": "hash", # default setting -- HASH
    },
    "fields": [
        {"name": "user", "type": "tag"},
        {"name": "credit_score", "type": "tag"},
        {"name": "job", "type": "text"},
        {"name": "age", "type": "numeric"},
        {"name": "office_location", "type": "geo"},
        {
            "name": "user_embedding",
            "type": "vector",
            "attrs": {
                "dims": 3,
                "distance_metric": "cosine",
                "algorithm": "flat",
                "datatype": "float32"
            }
        }
    ]
}
```

```
# construct a search index from the hash schema
hindex = SearchIndex.from_dict(hash_schema, redis_url="redis://localhost:6379")

# create the index (no data yet)
hindex.create(overwrite=True)
```

```
# show the underlying storage type
hindex.storage_type
```

```
<StorageType.HASH: 'hash'>
```

Vectors as byte strings

One nuance when working with Hashes in Redis, is that all vectorized data must be passed as a byte string (for efficient storage, indexing, and processing). An example of that can be seen below:

```
# show a single entry from the data that will be loaded
data[0]
```

```
{'user': 'john',
 'age': 18,
 'job': 'engineer',
```

(continues on next page)

(continued from previous page)

```
'credit_score': 'high',
'office_location': '-122.4194,37.7749',
'user_embedding': b'\xcd\xcc\xcc=\xcd\xcc\xcc=\x00\x00\x00?',
'last_updated': 1741627789}
```

```
# load hash data
keys = hindex.load(data)
```

```
!rvl stats -i user-hash
```

Statistics:

Stat Key	Value
num_docs	7
num_terms	6
max_doc_id	7
num_records	44
percent_indexed	1
hash_indexing_failures	0
number_of_uses	1
bytes_per_record_avg	40.2954559
doc_table_size_mb	7.27653503
inverted_sz_mb	0.00169086
key_table_size_mb	2.48908996
offset_bits_per_record_avg	8
offset_vectors_sz_mb	8.58306884
offsets_per_term_avg	0.20454545
records_per_doc_avg	6.28571414
sortable_values_size_mb	0
total_indexing_time	0.25799998
total_inverted_index_blocks	18
vector_index_sz_mb	0.02023315

Performing Queries

Once our index is created and data is loaded into the right format, we can run queries against the index with RedisVL:

```
from redisvl.query import VectorQuery
from redisvl.query.filter import Tag, Text, Num

t = (Tag("credit_score") == "high") & (Text("job") % "enginee*") & (Num("age") > 17) #_
↳ codespell:ignore enginee

v = VectorQuery(
    vector=[0.1, 0.1, 0.5],
    vector_field_name="user_embedding",
    return_fields=["user", "credit_score", "age", "job", "office_location"],
    filter_expression=t
```

(continues on next page)

(continued from previous page)

```
)

results = hindex.query(v)
result_print(results)
```

```
<IPython.core.display.HTML object>
```

```
# clean up
hindex.delete()
```

Working with JSON

JSON is best suited for use cases with the following characteristics:

- Ease of use and data model flexibility are top concerns
- Application data is already native JSON
- Replacing another document storage/db solution

```
# define the json index schema
json_schema = {
    "index": {
        "name": "user-json",
        "prefix": "user-json-docs",
        "storage_type": "json", # JSON storage type
    },
    "fields": [
        {"name": "user", "type": "tag"},
        {"name": "credit_score", "type": "tag"},
        {"name": "job", "type": "text"},
        {"name": "age", "type": "numeric"},
        {"name": "office_location", "type": "geo"},
        {
            "name": "user_embedding",
            "type": "vector",
            "attrs": {
                "dims": 3,
                "distance_metric": "cosine",
                "algorithm": "flat",
                "datatype": "float32"
            }
        }
    ]
}
```

```
# construct a search index from the json schema
jindex = SearchIndex.from_dict(json_schema, redis_url="redis://localhost:6379")
```

(continues on next page)

(continued from previous page)

```
# create the index (no data yet)
jindex.create(overwrite=True)
```

```
# note the multiple indices in the same database
!rvm index listall
```

```
13:02:56 [RedisVL] INFO    Indices:
13:02:56 [RedisVL] INFO    1. user-json
```

Vectors as float arrays

Vectorized data stored in JSON must be stored as a pure array (python list) of floats. We will modify our sample data to account for this below:

```
json_data = data.copy()

for d in json_data:
    d['user_embedding'] = buffer_to_array(d['user_embedding'], dtype='float32')
```

```
# inspect a single JSON record
json_data[0]
```

```
{'user': 'john',
 'age': 18,
 'job': 'engineer',
 'credit_score': 'high',
 'office_location': '-122.4194,37.7749',
 'user_embedding': [0.10000000149011612, 0.10000000149011612, 0.5],
 'last_updated': 1741627789}
```

```
keys = jindex.load(json_data)
```

```
# we can now run the exact same query as above
result_print(jindex.query(v))
```

```
<IPython.core.display.HTML object>
```

Cleanup

```
jindex.delete()
```

2.3.6 Working with nested data in JSON

Redis also supports native **JSON** objects. These can be multi-level (nested) objects, with full JSONPath support for updating/retrieving sub elements:

```
{
  "name": "Specialized Stump jumper",
  "metadata": {
    "model": "Stumpjumper",
    "brand": "Specialized",
    "type": "Enduro bikes",
    "price": 3000
  },
}
```

Full JSON Path support

Because Redis enables full JSON path support, when creating an index schema, elements need to be indexed and selected by their path with the desired name AND path that points to where the data is located within the objects.

By default, RedisVL will assume the path as `$.{name}` if not provided in JSON fields schema. If nested provide path as `$.object.attribute`

As an example:

```
from redisvl.utils.vectorize import HFTextVectorizer

emb_model = HFTextVectorizer()

bike_data = [
  {
    "name": "Specialized Stump jumper",
    "metadata": {
      "model": "Stumpjumper",
      "brand": "Specialized",
      "type": "Enduro bikes",
      "price": 3000
    },
    "description": "The Specialized Stumpjumper is a versatile enduro bike that
↳ dominates both climbs and descents. Features a FACT 11m carbon fiber frame, FOX FLOAT
↳ suspension with 160mm travel, and SRAM X01 Eagle drivetrain. The asymmetric frame
↳ design and internal storage compartment make it a practical choice for all-day
↳ adventures."
  },
  {
    "name": "bike_2",
    "metadata": {
      "model": "Slash",
      "brand": "Trek",
      "type": "Enduro bikes",
      "price": 5000
    },
  },
]
```

(continues on next page)

(continued from previous page)

```

        "description": "Trek's Slash is built for aggressive enduro riding and racing.
        ↳ Featuring Trek's Alpha Aluminum frame with RE:aktiv suspension technology, 160mm
        ↳ travel, and Knock Block frame protection. Equipped with Bontrager components and a
        ↳ Shimano XT drivetrain, this bike excels on technical trails and enduro race courses."
    }
]

bike_data = [{**d, "bike_embedding": emb_model.embed(d["description"])} for d in bike_
↳ data]

bike_schema = {
    "index": {
        "name": "bike-json",
        "prefix": "bike-json",
        "storage_type": "json", # JSON storage type
    },
    "fields": [
        {
            "name": "model",
            "type": "tag",
            "path": "$.metadata.model" # note the '$'
        },
        {
            "name": "brand",
            "type": "tag",
            "path": "$.metadata.brand"
        },
        {
            "name": "price",
            "type": "numeric",
            "path": "$.metadata.price"
        },
        {
            "name": "bike_embedding",
            "type": "vector",
            "attrs": {
                "dims": len(bike_data[0]["bike_embedding"]),
                "distance_metric": "cosine",
                "algorithm": "flat",
                "datatype": "float32"
            }
        }
    ],
}

```

```

/Users/tyler.hutcherson/Documents/AppliedAI/redis-vl-python/.venv/lib/python3.13/site-
↳ packages/tqdm/auto.py:21: TqdmWarning: IPProgress not found. Please update jupyter and
↳ ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
from .autonotebook import tqdm as notebook_tqdm

```

```
13:02:58 sentence_transformers.SentenceTransformer INFO    Use pytorch device_name: mps
13:02:58 sentence_transformers.SentenceTransformer INFO    Load pretrained_
↳ SentenceTransformer: sentence-transformers/all-mpnet-base-v2
```

```
Batches: 100%| 1/1 [00:00<00:00, 7.23it/s]
Batches: 100%| 1/1 [00:00<00:00, 12.93it/s]
Batches: 100%| 1/1 [00:00<00:00, 14.10it/s]
```

```
# construct a search index from the json schema
bike_index = SearchIndex.from_dict(bike_schema, redis_url="redis://localhost:6379")

# create the index (no data yet)
bike_index.create(overwrite=True)
```

```
bike_index.load(bike_data)
```

```
['bike-json:01JY4J9M48CXF7F4Y6HRGEMT9B',
 'bike-json:01JY4J9M48RRY6F80HR82CVZ5G']
```

```
from redisvl.query import VectorQuery

vec = emb_model.embed("I'd like a bike for aggressive riding")

v = VectorQuery(
    vector=vec,
    vector_field_name="bike_embedding",
    return_fields=[
        "brand",
        "name",
        "$.metadata.type"
    ]
)

results = bike_index.query(v)
```

```
Batches: 100%| 1/1 [00:00<00:00, 11.72it/s]
```

Note: As shown in the example if you want to retrieve a field from json object that was not indexed you will also need to supply the full path as with `$.metadata.type`.

```
results
```

```
[{'id': 'bike-json:01JY4J9M48RRY6F80HR82CVZ5G',
  'vector_distance': '0.519989132881',
  'brand': 'Trek',
  '$.metadata.type': 'Enduro bikes'},
 {'id': 'bike-json:01JY4J9M48CXF7F4Y6HRGEMT9B',
  'vector_distance': '0.657624304295',
  'brand': 'Specialized',
  '$.metadata.type': 'Enduro bikes'}]
```

```
# Cleanup
bike_index.delete()
```

2.3.7 Rerankers

In this notebook, we will show how to use RedisVL to rerank search results (documents or chunks or records) based on the input query. Today RedisVL supports reranking through:

- A re-ranker that uses pre-trained [Cross-Encoders](#) which can use models from [Hugging Face cross encoder models](#) or Hugging Face models that implement a cross encoder function (example: [BAAI/bge-reranker-base](#)).
- The [Cohere /rerank API](#).
- The [VoyageAI /rerank API](#).

Before running this notebook, be sure to:

1. Have installed `redisvl` and have that environment active for this notebook.
2. Have a running Redis Stack instance with RediSearch > 2.4 active.

For example, you can run Redis Stack locally with Docker:

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

This will run Redis on port 6379 and RedisInsight at <http://localhost:8001>.

```
# import necessary modules
import os
```

Simple Reranking

Reranking provides a relevance boost to search results generated by traditional (lexical) or semantic search strategies.

As a simple demonstration, take the passages and user query below:

```
query = "What is the capital of the United States?"
docs = [
    "Carson City is the capital city of the American state of Nevada. At the 2010 United_
↪ States Census, Carson City had a population of 55,274.",
    "The Commonwealth of the Northern Mariana Islands is a group of islands in the_
↪ Pacific Ocean that are a political division controlled by the United States. Its_
↪ capital is Saipan.",
    "Charlotte Amalie is the capital and largest city of the United States Virgin_
↪ Islands. It has about 20,000 people. The city is on the island of Saint Thomas.",
    "Washington, D.C. (also known as simply Washington or D.C., and officially as the_
↪ District of Columbia) is the capital of the United States. It is a federal district._
↪ The President of the USA and many major national government offices are in the_
↪ territory. This makes it the political center of the United States of America.",
    "Capital punishment (the death penalty) has existed in the United States since_
↪ before the United States was a country. As of 2017, capital punishment is legal in 30_
↪ of the 50 states. The federal government (including the United States military) also_
↪ uses capital punishment."
]
```

The goal of reranking is to provide a more fine-grained quality improvement to initial search results. With RedisVL, this would likely be results coming back from a search operation like full text or vector.

Using the Cross-Encoder Reranker

To use the cross-encoder reranker we initialize an instance of `HFCrossEncoderReranker` passing a suitable model (if no model is provided, the `cross-encoder/ms-marco-MiniLM-L-6-v2` model is used):

```
from redisvl.utils.rerank import HFCrossEncoderReranker

cross_encoder_reranker = HFCrossEncoderReranker("BAAI/bge-reranker-base")
```

Rerank documents with HFCrossEncoderReranker

With the obtained reranker instance we can rerank and truncate the list of documents based on relevance to the initial query.

```
results, scores = cross_encoder_reranker.rank(query=query, docs=docs)
```

```
for result, score in zip(results, scores):
    print(score, " -- ", result)
```

```
0.07461125403642654 -- {'content': 'Washington, D.C. (also known as simply Washington,
↳ or D.C., and officially as the District of Columbia) is the capital of the United
↳ States. It is a federal district. The President of the USA and many major national
↳ government offices are in the territory. This makes it the political center of the
↳ United States of America.'}
0.05220315232872963 -- {'content': 'Charlotte Amalie is the capital and largest city
↳ of the United States Virgin Islands. It has about 20,000 people. The city is on the
↳ island of Saint Thomas.'}
0.3802368640899658 -- {'content': 'Carson City is the capital city of the American
↳ state of Nevada. At the 2010 United States Census, Carson City had a population of 55,
↳ 274.'}
```

Using the Cohere Reranker

To initialize the Cohere reranker you'll need to install the cohere library and provide the right Cohere API Key.

```
#!/pip install cohere
```

```
import getpass

# setup the API Key
api_key = os.environ.get("COHERE_API_KEY") or getpass.getpass("Enter your Cohere API key:
↳ ")
```

```
from redisvl.utils.rerank import CohereReranker

cohere_reranker = CohereReranker(limit=3, api_config={"api_key": api_key})
```

Rerank documents with CohereReranker

Below we will use the CohereReranker to rerank and truncate the list of documents above based on relevance to the initial query.

```
results, scores = cohere_reranker.rank(query=query, docs=docs)
```

```
for result, score in zip(results, scores):
    print(score, " -- ", result)
```

```
0.9990564 -- Washington, D.C. (also known as simply Washington or D.C., and officially
↳as the District of Columbia) is the capital of the United States. It is a federal
↳district. The President of the USA and many major national government offices are in
↳the territory. This makes it the political center of the United States of America.
0.7516481 -- Capital punishment (the death penalty) has existed in the United States
↳since before the United States was a country. As of 2017, capital punishment is legal
↳in 30 of the 50 states. The federal government (including the United States military)
↳also uses capital punishment.
0.08882029 -- The Commonwealth of the Northern Mariana Islands is a group of islands
↳in the Pacific Ocean that are a political division controlled by the United States.
↳Its capital is Saipan.
```

Working with semi-structured documents

Often times the initial result set includes other metadata and components that could be used to steer the reranking relevancy. To accomplish this, we can set the `rank_by` argument and provide documents with those additional fields.

```
docs = [
    {
        "source": "wiki",
        "passage": "Carson City is the capital city of the American state of Nevada. At
↳the 2010 United States Census, Carson City had a population of 55,274."
    },
    {
        "source": "encyclopedia",
        "passage": "The Commonwealth of the Northern Mariana Islands is a group of
↳islands in the Pacific Ocean that are a political division controlled by the United
↳States. Its capital is Saipan."
    },
    {
        "source": "textbook",
        "passage": "Charlotte Amalie is the capital and largest city of the United
↳States Virgin Islands. It has about 20,000 people. The city is on the island of Saint
↳Thomas."
    },
    {
        "source": "textbook",
        "passage": "Washington, D.C. (also known as simply Washington or D.C., and
↳officially as the District of Columbia) is the capital of the United States. It is a
↳federal district. The President of the USA and many major national government offices
↳are in the territory. This makes it the political center of the United States of
```

(continues on next page)

(continued from previous page)

```

    ↪America."
    },
    {
        "source": "wiki",
        "passage": "Capital punishment (the death penalty) has existed in the United
    ↪States since before the United States was a country. As of 2017, capital punishment is
    ↪legal in 30 of the 50 states. The federal government (including the United States
    ↪military) also uses capital punishment."
    }
]

```

```

results, scores = cohere_reranker.rank(query=query, docs=docs, rank_by=["passage",
    ↪"source"])

```

```

for result, score in zip(results, scores):
    print(score, " -- ", result)

```

```

0.9988121 -- {'source': 'textbook', 'passage': 'Washington, D.C. (also known as simply
    ↪Washington or D.C., and officially as the District of Columbia) is the capital of the
    ↪United States. It is a federal district. The President of the USA and many major
    ↪national government offices are in the territory. This makes it the political center
    ↪of the United States of America.'}
0.5974905 -- {'source': 'wiki', 'passage': 'Capital punishment (the death penalty) has
    ↪existed in the United States since before the United States was a country. As of 2017,
    ↪capital punishment is legal in 30 of the 50 states. The federal government (including
    ↪the United States military) also uses capital punishment.'}
0.059101548 -- {'source': 'encyclopedia', 'passage': 'The Commonwealth of the Northern
    ↪Mariana Islands is a group of islands in the Pacific Ocean that are a political
    ↪division controlled by the United States. Its capital is Saipan.'}

```

Using the VoyageAI Reranker

To initialize the VoyageAI reranker you'll need to install the voyaeai library and provide the right VoyageAI API Key.

```

#!pip install voyageai

```

```

import getpass

# setup the API Key
api_key = os.environ.get("VOYAGE_API_KEY") or getpass.getpass("Enter your VoyageAI API
    ↪key: ")

```

```

from redisvl.utils.rerank import VoyageAIReranker

reranker = VoyageAIReranker(model="rerank-lite-1", limit=3, api_config={"api_key": api_
    ↪key})# Please check the available models at https://docs.voyageai.com/docs/reranker

```

Rerank documents with VoyageAIReranker

Below we will use the VoyageAIReranker to rerank and also truncate the list of documents above based on relevance to the initial query.

```
results, scores = reranker.rank(query=query, docs=docs)
```

```
for result, score in zip(results, scores):  
    print(score, " -- ", result)
```

```
0.796875 -- Washington, D.C. (also known as simply Washington or D.C., and officially,  
↳as the District of Columbia) is the capital of the United States. It is a federal,  
↳district. The President of the USA and many major national government offices are in,  
↳the territory. This makes it the political center of the United States of America.  
0.578125 -- Charlotte Amalie is the capital and largest city of the United States,  
↳Virgin Islands. It has about 20,000 people. The city is on the island of Saint Thomas.  
0.5625 -- Carson City is the capital city of the American state of Nevada. At the 2010,  
↳United States Census, Carson City had a population of 55,274.
```

2.3.8 LLM Message History

Large Language Models are inherently stateless and have no knowledge of previous interactions with a user, or even of previous parts of the current conversation. While this may not be noticeable when asking simple questions, it becomes a hindrance when engaging in long running conversations that rely on conversational context.

The solution to this problem is to append the previous conversation history to each subsequent call to the LLM.

This notebook will show how to use Redis to structure and store and retrieve this conversational message history.

```
from redisvl.extensions.message_history import MessageHistory
```

```
chat_history = MessageHistory(name='student tutor')
```

To align with common LLM APIs, Redis stores messages with role and content fields. The supported roles are “system”, “user” and “llm”.

You can store messages one at a time or all at once.

```
chat_history.add_message({"role": "system", "content": "You are a helpful geography tutor,  
↳giving simple and short answers to questions about European countries."})  
chat_history.add_messages([  
    {"role": "user", "content": "What is the capital of France?"},  
    {"role": "llm", "content": "The capital is Paris."},  
    {"role": "user", "content": "And what is the capital of Spain?"},  
    {"role": "llm", "content": "The capital is Madrid."},  
    {"role": "user", "content": "What is the population of Great Britain?"},  
    {"role": "llm", "content": "As of 2023 the population of Great Britain is,  
↳approximately 67 million people."},  
)
```

At any point we can retrieve the recent history of the conversation. It will be ordered by entry time.

```
context = chat_history.get_recent()
for message in context:
    print(message)
```

```
{'role': 'llm', 'content': 'The capital is Paris.'}
{'role': 'user', 'content': 'And what is the capital of Spain?'}
{'role': 'llm', 'content': 'The capital is Madrid.'}
{'role': 'user', 'content': 'What is the population of Great Britain?'}
{'role': 'llm', 'content': 'As of 2023 the population of Great Britain is approximately
↳67 million people.'}
```

In many LLM flows the conversation progresses in a series of prompt and response pairs. Message history offer a convenience function `store()` to add these simply.

```
prompt = "what is the size of England compared to Portugal?"
response = "England is larger in land area than Portal by about 15000 square miles."
chat_history.store(prompt, response)
```

```
context = chat_history.get_recent(top_k=6)
for message in context:
    print(message)
```

```
{'role': 'user', 'content': 'And what is the capital of Spain?'}
{'role': 'llm', 'content': 'The capital is Madrid.'}
{'role': 'user', 'content': 'What is the population of Great Britain?'}
{'role': 'llm', 'content': 'As of 2023 the population of Great Britain is approximately
↳67 million people.'}
{'role': 'user', 'content': 'what is the size of England compared to Portugal?'}
{'role': 'llm', 'content': 'England is larger in land area than Portal by about 15000
↳square miles.'}
```

Managing multiple users and conversations

For applications that need to handle multiple conversations concurrently, Redis supports tagging messages to keep conversations separated.

```
chat_history.add_message({"role": "system", "content": "You are a helpful algebra tutor,
↳giving simple answers to math problems."}, session_tag='student two')
chat_history.add_messages([
    {"role": "user", "content": "What is the value of x in the equation 2x + 3 = 7?"},
    {"role": "llm", "content": "The value of x is 2."},
    {"role": "user", "content": "What is the value of y in the equation 3y - 5 = 7?"},
    {"role": "llm", "content": "The value of y is 4."}],
    session_tag='student two'
)

for math_message in chat_history.get_recent(session_tag='student two'):
    print(math_message)
```

```
{'role': 'system', 'content': 'You are a helpful algebra tutor, giving simple answers to
↳math problems.'}
```

(continues on next page)

(continued from previous page)

```
{'role': 'user', 'content': 'What is the value of x in the equation 2x + 3 = 7?'}
{'role': 'llm', 'content': 'The value of x is 2.'}
{'role': 'user', 'content': 'What is the value of y in the equation 3y - 5 = 7?'}
{'role': 'llm', 'content': 'The value of y is 4.'}
```

Semantic message history

For longer conversations our list of messages keeps growing. Since LLMs are stateless we have to continue to pass this conversation history on each subsequent call to ensure the LLM has the correct context.

A typical flow looks like this:

```
while True:
    prompt = input('enter your next question')
    context = chat_history.get_recent()
    response = LLM_api_call(prompt=prompt, context=context)
    chat_history.store(prompt, response)
```

This works, but as context keeps growing so too does our LLM token count, which increases latency and cost.

Conversation histories can be truncated, but that can lead to losing relevant information that appeared early on.

A better solution is to pass only the relevant conversational context on each subsequent call.

For this, RedisVL has the `SemanticMessageHistory`, which uses vector similarity search to return only semantically relevant sections of the conversation.

```
from redisvl.extensions.message_history import SemanticMessageHistory
semantic_history = SemanticMessageHistory(name='tutor')

semantic_history.add_messages(chat_history.get_recent(top_k=8))
```

```
/Users/tyler.hutcherson/Documents/AppliedAI/redis-vl-python/.venv/lib/python3.13/site-
packages/tqdm/auto.py:21: TqdmWarning: IPProgress not found. Please update jupyter and
ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
from .autonotebook import tqdm as notebook_tqdm
```

```
13:03:39 sentence_transformers.SentenceTransformer INFO    Use pytorch device_name: mps
13:03:39 sentence_transformers.SentenceTransformer INFO    Load pretrained
SentenceTransformer: sentence-transformers/all-mpnet-base-v2
```

```
Batches: 100%| 1/1 [00:00<00:00, 6.59it/s]
Batches: 100%| 1/1 [00:00<00:00, 10.33it/s]
Batches: 100%| 1/1 [00:00<00:00, 9.91it/s]
Batches: 100%| 1/1 [00:00<00:00, 12.52it/s]
Batches: 100%| 1/1 [00:00<00:00, 57.92it/s]
Batches: 100%| 1/1 [00:00<00:00, 60.45it/s]
Batches: 100%| 1/1 [00:00<00:00, 13.38it/s]
Batches: 100%| 1/1 [00:00<00:00, 13.65it/s]
Batches: 100%| 1/1 [00:00<00:00, 62.33it/s]
```

```
prompt = "what have I learned about the size of England?"
semantic_history.set_distance_threshold(0.35)
context = semantic_history.get_relevant(prompt)
for message in context:
    print(message)
```

```
Batches: 100%|| 1/1 [00:00<00:00, 56.30it/s]
```

```
{'role': 'user', 'content': 'what is the size of England compared to Portugal?'}
```

You can adjust the degree of semantic similarity needed to be included in your context.

Setting a distance threshold close to 0.0 will require an exact semantic match, while a distance threshold of 1.0 will include everything.

```
semantic_history.set_distance_threshold(0.7)

larger_context = semantic_history.get_relevant(prompt)
for message in larger_context:
    print(message)
```

```
Batches: 100%|| 1/1 [00:00<00:00, 50.04it/s]
```

```
{'role': 'user', 'content': 'what is the size of England compared to Portugal?'}
{'role': 'llm', 'content': 'England is larger in land area than Portal by about 15000_
↳square miles.'}
{'role': 'user', 'content': 'What is the population of Great Britain?'}
{'role': 'llm', 'content': 'As of 2023 the population of Great Britain is approximately_
↳67 million people.'}
{'role': 'user', 'content': 'And what is the capital of Spain?'}
```

Conversation control

LLMs can hallucinate on occasion and when this happens it can be useful to prune incorrect information from conversational histories so this incorrect information doesn't continue to be passed as context.

```
semantic_history.store(
    prompt="what is the smallest country in Europe?",
    response="Monaco is the smallest country in Europe at 0.78 square miles." #_
↳Incorrect. Vatican City is the smallest country in Europe
)

# get the key of the incorrect message
context = semantic_history.get_recent(top_k=1, raw=True)
bad_key = context[0]['entry_id']
semantic_history.drop(bad_key)
```

(continues on next page)

(continued from previous page)

```
corrected_context = semantic_history.get_recent()
for message in corrected_context:
    print(message)
```

```
Batches: 100%|| 1/1 [00:00<00:00, 54.73it/s]
Batches: 100%|| 1/1 [00:00<00:00, 10.63it/s]
```

```
{'role': 'user', 'content': 'What is the population of Great Britain?'}
{'role': 'llm', 'content': 'As of 2023 the population of Great Britain is approximately
↪67 million people.'}
{'role': 'user', 'content': 'what is the size of England compared to Portugal?'}
{'role': 'llm', 'content': 'England is larger in land area than Portal by about 15000
↪square miles.'}
{'role': 'user', 'content': 'what is the smallest country in Europe?'}
```

```
chat_history.clear()
semantic_history.clear()
```

2.3.9 Semantic Routing

RedisVL provides a `SemanticRouter` interface to utilize Redis' built-in search & aggregation in order to perform KNN-style classification over a set of Route references to determine the best match.

This notebook will go over how to use Redis as a Semantic Router for your applications

Define the Routes

Below we define 3 different routes. One for **technology**, one for **sports**, and another for **entertainment**. Now for this example, the goal here is surely topic “classification”. But you can create routes and references for almost anything.

Each route has a set of references that cover the “semantic surface area” of the route. The incoming query from a user needs to be semantically similar to one or more of the references in order to “match” on the route.

Additionally, each route has a `distance_threshold` which determines the maximum distance between the query and the reference for the query to be routed to the route. This value is unique to each route.

```
from redisvl.extensions.router import Route

# Define routes for the semantic router
technology = Route(
    name="technology",
    references=[
        "what are the latest advancements in AI?",
        "tell me about the newest gadgets",
        "what's trending in tech?"
    ],
    metadata={"category": "tech", "priority": 1},
    distance_threshold=0.71
```

(continues on next page)

(continued from previous page)

```

)

sports = Route(
    name="sports",
    references=[
        "who won the game last night?",
        "tell me about the upcoming sports events",
        "what's the latest in the world of sports?",
        "sports",
        "basketball and football"
    ],
    metadata={"category": "sports", "priority": 2},
    distance_threshold=0.72
)

entertainment = Route(
    name="entertainment",
    references=[
        "what are the top movies right now?",
        "who won the best actor award?",
        "what's new in the entertainment industry?"
    ],
    metadata={"category": "entertainment", "priority": 3},
    distance_threshold=0.7
)

```

Initialize the SemanticRouter

SemanticRouter will automatically create an index within Redis upon initialization for the route references. By default, it uses the HFTextVectorizer to generate embeddings for each route reference.

```

import os
from redisvl.extensions.router import SemanticRouter
from redisvl.utils.vectorize import HFTextVectorizer

os.environ["TOKENIZERS_PARALLELISM"] = "false"

# Initialize the SemanticRouter
router = SemanticRouter(
    name="topic-router",
    vectorizer=HFTextVectorizer(),
    routes=[technology, sports, entertainment],
    redis_url="redis://localhost:6379",
    overwrite=True # Blow away any other routing index with this name
)

```

```

/Users/tyler.hutcherson/Documents/AppliedAI/redis-vl-python/.venv/lib/python3.13/site-
packages/tqdm/auto.py:21: TqdmWarning: IPProgress not found. Please update jupyter and
ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
from .autonotebook import tqdm as notebook_tqdm

```

```
13:03:49 sentence_transformers.SentenceTransformer INFO    Use pytorch device_name: mps
13:03:49 sentence_transformers.SentenceTransformer INFO    Load pretrained_
↳SentenceTransformer: sentence-transformers/all-mpnet-base-v2
```

```
Batches: 100%| 1/1 [00:00<00:00, 6.31it/s]
Batches: 100%| 1/1 [00:00<00:00, 7.02it/s]
Batches: 100%| 1/1 [00:00<00:00, 8.21it/s]
Batches: 100%| 1/1 [00:00<00:00, 54.33it/s]
```

```
# look at the index specification created for the semantic router
!rvl index info -i topic-router
```

Index Information:				
Index Name	Storage Type	Prefixes	Index Options	Indexing
topic-router	HASH	['topic-router']	[]	0
Index Fields:				
Name	Attribute	Type	Field Option	Option Value
Field Option	Option Value	Field Option	Option Value	Field_
Option	Option Value			
reference_id	reference_id	TAG	SEPARATOR	,
route_name	route_name	TAG	SEPARATOR	,
reference	reference	TEXT	WEIGHT	1
vector	vector	VECTOR	algorithm	FLAT
data_type	FLOAT32	dim	768	distance_
metric	COSINE			

```
router._index.info()["num_docs"]
```

11

Simple routing

```
# Query the router with a statement
route_match = router("Can you tell me about the latest in artificial intelligence?")
route_match
```

```
Batches: 100%| 1/1 [00:00<00:00, 8.63it/s]
```

```
RouteMatch(name='technology', distance=0.419145941734)
```

```
# Query the router with a statement and return a miss
route_match = router("are aliens real?")
route_match
```

```
Batches: 100%| 1/1 [00:00<00:00, 11.71it/s]
```

```
RouteMatch(name=None, distance=None)
```

We can also route a statement to many routes and order them by distance:

```
# Perform multi-class classification with route_many() -- toggle the max_k and the
↪ distance_threshold
route_matches = router.route_many("How is AI used in basketball?", max_k=3)
route_matches
```

```
Batches: 100%| 1/1 [00:00<00:00, 12.12it/s]
```

```
[RouteMatch(name='technology', distance=0.556493639946),
 RouteMatch(name='sports', distance=0.671060085297)]
```

```
# Toggle the aggregation method -- note the different distances in the result
from redisvl.extensions.router.schema import DistanceAggregationMethod

route_matches = router.route_many("How is AI used in basketball?", aggregation_
↪ method=DistanceAggregationMethod.min, max_k=3)
route_matches
```

```
Batches: 100%| 1/1 [00:00<00:00, 56.69it/s]
```

```
[RouteMatch(name='technology', distance=0.556493639946),
 RouteMatch(name='sports', distance=0.629264354706)]
```

Note the different route match distances. This is because we used the min aggregation method instead of the default avg approach.

Update the routing config

```
from redisvl.extensions.router import RoutingConfig

router.update_routing_config(
    RoutingConfig(aggregation_method=DistanceAggregationMethod.min, max_k=3)
)
```

```
route_matches = router.route_many("Lebron James")
route_matches
```

```
Batches: 100%|| 1/1 [00:00<00:00, 13.20it/s]
```

```
[RouteMatch(name='sports', distance=0.663253903389)]
```

Router serialization

```
router.to_dict()
```

```
{'name': 'topic-router',
 'routes': [{'name': 'technology',
  'references': ['what are the latest advancements in AI?',
  'tell me about the newest gadgets',
  "what's trending in tech?"],
  'metadata': {'category': 'tech', 'priority': 1},
  'distance_threshold': 0.71},
 {'name': 'sports',
  'references': ['who won the game last night?',
  'tell me about the upcoming sports events',
  "what's the latest in the world of sports?",
  'sports',
  'basketball and football'],
  'metadata': {'category': 'sports', 'priority': 2},
  'distance_threshold': 0.72},
 {'name': 'entertainment',
  'references': ['what are the top movies right now?',
  'who won the best actor award?',
  "what's new in the entertainment industry?"],
  'metadata': {'category': 'entertainment', 'priority': 3},
  'distance_threshold': 0.7}],
 'vectorizer': {'type': 'hf',
  'model': 'sentence-transformers/all-mpnet-base-v2'},
 'routing_config': {'max_k': 3, 'aggregation_method': 'min'}}
```

```
router2 = SemanticRouter.from_dict(router.to_dict(), redis_url="redis://localhost:6379")

assert router2.to_dict() == router.to_dict()
```

```
13:03:54 sentence_transformers.SentenceTransformer INFO    Use pytorch device_name: mps
13:03:54 sentence_transformers.SentenceTransformer INFO    Load pretrained_
↪SentenceTransformer: sentence-transformers/all-mpnet-base-v2
```

```
Batches: 100%|| 1/1 [00:00<00:00, 53.91it/s]
```

```
13:03:54 redisvl.index.index INFO    Index already exists, not overwriting.
```

```
router.to_yaml("router.yaml", overwrite=True)
```

```
router3 = SemanticRouter.from_yaml("router.yaml", redis_url="redis://localhost:6379")
assert router3.to_dict() == router2.to_dict() == router.to_dict()
```

```
13:03:54 sentence_transformers.SentenceTransformer INFO    Use pytorch device_name: mps
13:03:54 sentence_transformers.SentenceTransformer INFO    Load pretrained_
↪SentenceTransformer: sentence-transformers/all-mpnet-base-v2
```

```
Batches: 100%|| 1/1 [00:00<00:00, 51.94it/s]
```

```
13:03:55 redisvl.index.index INFO    Index already exists, not overwriting.
```

Add route references

```
router.add_route_references(route_name="technology", references=["latest AI trends",
↪"new tech gadgets"])
```

```
Batches: 100%|| 1/1 [00:00<00:00, 8.12it/s]
```

```
['topic-router:technology:
↪f243fb2d073774e81c7815247cb3013794e6225df3cbe3769cee8c6cefaca777',
 'topic-router:technology:
↪7e4bca5853c1c3298b4d001de13c3c7a79a6e0f134f81acc2e7cddb6845961f']
```

Get route references

```
# by route name
refs = router.get_route_references(route_name="technology")
refs
```

```
[{'id': 'topic-router:technology:
↪85cc73a1437df27caa2f075a29c497e5a2e532023fbb75378aedbae80779ab37',
```

(continues on next page)

(continued from previous page)

```

    'reference_id': '85cc73a1437df27caa2f075a29c497e5a2e532023fbb75378aedbae80779ab37',
    'route_name': 'technology',
    'reference': 'tell me about the newest gadgets'},
    {'id': 'topic-router:technology:
    ↳851f51cce5a9ccfbcb66993908be6b7871479af3e3a4b139ad292a1bf7e0676',
    'reference_id': '851f51cce5a9ccfbcb66993908be6b7871479af3e3a4b139ad292a1bf7e0676',
    'route_name': 'technology',
    'reference': 'what are the latest advancements in AI?'},
    {'id': 'topic-router:technology:
    ↳f243fb2d073774e81c7815247cb3013794e6225df3cbe3769cee8c6cefaca777',
    'reference_id': 'f243fb2d073774e81c7815247cb3013794e6225df3cbe3769cee8c6cefaca777',
    'route_name': 'technology',
    'reference': 'latest AI trends'},
    {'id': 'topic-router:technology:
    ↳7e4bca5853c1c3298b4d001de13c3c7a79a6e0f134f81acc2e7cddb6845961f',
    'reference_id': '7e4bca5853c1c3298b4d001de13c3c7a79a6e0f134f81acc2e7cddb6845961f',
    'route_name': 'technology',
    'reference': 'new tech gadgets'},
    {'id': 'topic-router:technology:
    ↳149a9c9919c58534aa0f369e85ad95ba7f00aa0513e0f81e2aff2ea4a717b0e0',
    'reference_id': '149a9c9919c58534aa0f369e85ad95ba7f00aa0513e0f81e2aff2ea4a717b0e0',
    'route_name': 'technology',
    'reference': "what's trending in tech?"]}

```

by reference id

```

refs = router.get_route_references(reference_ids=[refs[0]["reference_id"]])
refs

```

```

[{'id': 'topic-router:technology:
    ↳85cc73a1437df27caa2f075a29c497e5a2e532023fbb75378aedbae80779ab37',
    'reference_id': '85cc73a1437df27caa2f075a29c497e5a2e532023fbb75378aedbae80779ab37',
    'route_name': 'technology',
    'reference': 'tell me about the newest gadgets'}]

```

Delete route references

by route name

```

deleted_count = router.delete_route_references(route_name="sports")
deleted_count

```

5

by id

```

deleted_count = router.delete_route_references(reference_ids=[refs[0]["reference_id"]])
deleted_count

```

1

Clean up the router

```
# Use clear to flush all routes from the index
router.clear()
```

```
# Use delete to clear the index and remove it completely
router.delete()
```

2.3.10 SVS-VAMANA Vector Search

In this notebook, we will explore SVS-VAMANA (Scalable Vector Search with VAMANA graph algorithm), a graph-based vector search algorithm that is optimized to work with compression methods to reduce memory usage. It combines the Vamana graph algorithm with advanced compression techniques (LVQ and LeanVec) and is optimized for Intel hardware.

How it works

Vamana builds a single-layer proximity graph and prunes edges during construction based on tunable parameters, similar to HNSW but with a simpler structure. The compression methods apply per-vector normalization and scalar quantization, learning parameters directly from the data to enable fast, on-the-fly distance computations with SIMD-optimized layout Vector quantization and compression.

SVS-VAMANA offers:

- **Fast approximate nearest neighbor search** using graph-based algorithms
- **Vector compression** (LVQ, LeanVec) with up to 87.5% memory savings
- **Dimensionality reduction** (optional, with LeanVec)
- **Automatic performance optimization** through CompressionAdvisor

Use SVS-VAMANA when:

- Large datasets where memory is expensive
- Cloud deployments with memory-based pricing
- When 90-95% recall is acceptable
- High-dimensional vectors (>1024 dims) with LeanVec compression

Table of Contents

1. *Prerequisites*
2. *Quick Start with CompressionAdvisor*
3. *Creating an SVS-VAMANA Index*
4. *Loading Sample Data*
5. *Performing Vector Searches*
6. *Understanding Compression Types*
7. *Hybrid Queries with SVS-VAMANA*
8. *Performance Monitoring*
9. *Manual Configuration (Advanced)*
10. *Best Practices and Tips*

11. *Cleanup*

Prerequisites

Before running this notebook, ensure you have:

1. Installed `redisvl` and have that environment active for this notebook
2. A running Redis Stack instance with:
 - Redis \geq 8.2.0
 - RediSearch \geq 2.8.10

For example, you can run Redis Stack locally with Docker:

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

Note: SVS-VAMANA only supports FLOAT16 and FLOAT32 datatypes.

```
# Import necessary modules
import numpy as np
from redisvl.index import SearchIndex
from redisvl.query import VectorQuery
from redisvl.utils import CompressionAdvisor
from redisvl.redis.utils import array_to_buffer

# Set random seed for reproducible results
np.random.seed(42)
```

```
# Redis connection
REDIS_URL = "redis://localhost:6379"
```

Quick Start with CompressionAdvisor

The easiest way to get started with SVS-VAMANA is using the `CompressionAdvisor` utility, which automatically recommends optimal configuration based on your vector dimensions and performance priorities.

```
# Get recommended configuration for common embedding dimensions
dims = 1024 # Common embedding dimensions (works reliably with SVS-VAMANA)

config = CompressionAdvisor.recommend(
    dims=dims,
    priority="balanced" # Options: "memory", "speed", "balanced"
)

print("Recommended Configuration:")
for key, value in config.items():
    print(f" {key}: {value}")

# Estimate memory savings
savings = CompressionAdvisor.estimate_memory_savings(
```

(continues on next page)

(continued from previous page)

```

    config["compression"],
    dims,
    config.get("reduce")
)
print(f"\nEstimated Memory Savings: {savings}%")

```

Recommended Configuration:

```

algorithm: svsvamana
datatype: float16
graph_max_degree: 64
construction_window_size: 300
compression: LeanVec4x8
reduce: 512
search_window_size: 30

```

Estimated Memory Savings: 81.2%

Creating an SVS-VAMANA Index

Let's create an index using the recommended configuration. We'll use a simple schema with text content and vector embeddings.

```

# Create index schema with recommended SVS-VAMANA configuration
schema = {
    "index": {
        "name": "svs_demo",
        "prefix": "doc",
    },
    "fields": [
        {"name": "content", "type": "text"},
        {"name": "category", "type": "tag"},
        {
            "name": "embedding",
            "type": "vector",
            "attrs": {
                "dims": dims,
                **config, # Use the recommended configuration
                "distance_metric": "cosine"
            }
        }
    ]
}

# Create the index
index = SearchIndex.from_dict(schema, redis_url=REDIS_URL)
index.create(overwrite=True)

print(f" Created SVS-VAMANA index: {index.name}")
print(f"   Algorithm: {config['algorithm']}")
print(f"   Compression: {config['compression']}")
print(f"   Dimensions: {dims}")

```

(continues on next page)

(continued from previous page)

```
if 'reduce' in config:
    print(f"    Reduced to: {config['reduce']} dimensions")
```

```
Created SVS-VAMANA index: svd_demo
Algorithm: svd-vamana
Compression: LeanVec4x8
Dimensions: 1024
Reduced to: 512 dimensions
```

Loading Sample Data

Let's create some sample documents with embeddings to demonstrate SVS-VAMANA search capabilities.

```
# Generate sample data
sample_documents = [
    {"content": "Machine learning algorithms for data analysis", "category": "technology"},
    {"content": "Natural language processing and text understanding", "category": "technology"},
    {"content": "Computer vision and image recognition systems", "category": "technology"},
    {"content": "Delicious pasta recipes from Italy", "category": "food"},
    {"content": "Traditional French cooking techniques", "category": "food"},
    {"content": "Healthy meal planning and nutrition", "category": "food"},
    {"content": "Travel guide to European destinations", "category": "travel"},
    {"content": "Adventure hiking in mountain regions", "category": "travel"},
    {"content": "Cultural experiences in Asian cities", "category": "travel"},
    {"content": "Financial planning for retirement", "category": "finance"},
]

# Generate random embeddings for demonstration
# In practice, you would use a real embedding model
data_to_load = []

# Use reduced dimensions if LeanVec compression is applied
vector_dims = config.get("reduce", dims)
print(f"Creating vectors with {vector_dims} dimensions (reduced from {dims} if applicable)")

for i, doc in enumerate(sample_documents):
    # Create a random vector with some category-based clustering
    base_vector = np.random.random(vector_dims).astype(np.float32)

    # Add some category-based similarity (optional, for demo purposes)
    category_offset = hash(doc["category"]) % 100 / 1000.0
    base_vector[0] += category_offset

    # Convert to the datatype specified in config
    if config["datatype"] == "float16":
        base_vector = base_vector.astype(np.float16)
```

(continues on next page)

(continued from previous page)

```

data_to_load.append({
    "content": doc["content"],
    "category": doc["category"],
    "embedding": array_to_buffer(base_vector, dtype=config["datatype"])
})

# Load data into the index
index.load(data_to_load)
print(f" Loaded {len(data_to_load)} documents into the index")

# Wait a moment for indexing to complete
import time
time.sleep(2)

# Verify the data was loaded
info = index.info()
print(f"   Index now contains {info.get('num_docs', 0)} documents")

```

```

Creating vectors with 512 dimensions (reduced from 1024 if applicable)
Loaded 10 documents into the index
Index now contains 0 documents

```

Performing Vector Searches

Now let's perform some vector similarity searches using our SVS-VAMANA index.

```

# Create a query vector (in practice, this would be an embedding of your query text)
# Important: Query vector must match the index datatype and dimensions
vector_dims = config.get("reduce", dims)
if config["datatype"] == "float16":
    query_vector = np.random.random(vector_dims).astype(np.float16)
else:
    query_vector = np.random.random(vector_dims).astype(np.float32)

# Perform a vector similarity search
query = VectorQuery(
    vector=query_vector.tolist(),
    vector_field_name="embedding",
    return_fields=["content", "category"],
    num_results=5
)

results = index.query(query)

print(" Vector Search Results:")
print("=" * 50)
for i, result in enumerate(results, 1):
    distance = result.get('vector_distance', 'N/A')
    print(f"{i}. [{result['category']}] {result['content']}")
    print(f"   Distance: {distance:.4f}" if isinstance(distance, (int, float)) else f"   ")

```

(continues on next page)

(continued from previous page)

```
↪Distance: {distance}")
print()
```

Vector Search Results:

=====

Understanding Compression Types

SVS-VAMANA supports different compression algorithms that trade off between memory usage and search quality. Let's explore the available options.

```
# Compare different compression priorities
print("Compression Recommendations for Different Priorities:")
print("=" * 60)

priorities = ["memory", "speed", "balanced"]
for priority in priorities:
    config = CompressionAdvisor.recommend(dims=dims, priority=priority)
    savings = CompressionAdvisor.estimate_memory_savings(
        config["compression"],
        dims,
        config.get("reduce")
    )

    print(f"\n{priority.upper()} Priority:")
    print(f"  Compression: {config['compression']}")
    print(f"  Datatype: {config['datatype']}")
    if "reduce" in config:
        print(f"  Dimensionality reduction: {dims} → {config['reduce']}")
    print(f"  Search window size: {config['search_window_size']}")
    print(f"  Memory savings: {savings}%")
```

Compression Recommendations for Different Priorities:

=====

MEMORY Priority:

```
Compression: LeanVec4x8
Datatype: float16
Dimensionality reduction: 1024 → 512
Search window size: 20
Memory savings: 81.2%
```

SPEED Priority:

```
Compression: LeanVec4x8
Datatype: float16
Dimensionality reduction: 1024 → 256
Search window size: 40
Memory savings: 90.6%
```

BALANCED Priority:

(continues on next page)

(continued from previous page)

```

Compression: LeanVec4x8
Datatype: float16
Dimensionality reduction: 1024 → 512
Search window size: 30
Memory savings: 81.2%

```

Compression Types Explained

SVS-VAMANA offers several compression algorithms:

LVQ (Learned Vector Quantization)

- **LVQ4**: 4 bits per dimension (87.5% memory savings)
- **LVQ4x4**: 8 bits per dimension (75% memory savings)
- **LVQ4x8**: 12 bits per dimension (62.5% memory savings)
- **LVQ8**: 8 bits per dimension (75% memory savings)

LeanVec (Compression + Dimensionality Reduction)

- **LeanVec4x8**: 12 bits per dimension + dimensionality reduction
- **LeanVec8x8**: 16 bits per dimension + dimensionality reduction

The CompressionAdvisor automatically chooses the best compression type based on your vector dimensions and priority.

```

# Demonstrate compression savings for different vector dimensions
test_dimensions = [384, 768, 1024, 1536, 3072]

print("Memory Savings by Vector Dimension:")
print("=" * 50)
print(f"{'Dims':<6} {'Compression':<12} {'Savings':<8} {'Strategy'}")
print("-" * 50)

for dims in test_dimensions:
    config = CompressionAdvisor.recommend(dims=dims, priority="balanced")
    savings = CompressionAdvisor.estimate_memory_savings(
        config["compression"],
        dims,
        config.get("reduce")
    )

    strategy = "LeanVec" if dims >= 1024 else "LVQ"
    print(f"{'dims':<6} {'config['compression']':<12} {'savings:>6.1f}% {strategy}")

```

Memory Savings by Vector Dimension:

```

=====
Dims    Compression  Savings  Strategy

```

(continues on next page)

(continued from previous page)

```
-----
384    LVQ4x4      75.0% LVQ
768    LVQ4x4      75.0% LVQ
1024   LeanVec4x8  81.2% LeanVec
1536   LeanVec4x8  81.2% LeanVec
3072   LeanVec4x8  81.2% LeanVec
```

Hybrid Queries with SVS-VAMANA

SVS-VAMANA can be combined with other Redis search capabilities for powerful hybrid queries that filter by metadata while performing vector similarity search.

```
# Perform a hybrid search: vector similarity + category filter
hybrid_query = VectorQuery(
    vector=query_vector.tolist(),
    vector_field_name="embedding",
    return_fields=["content", "category"],
    num_results=3
)

# Add a filter to only search within "technology" category
hybrid_query.set_filter("@category:{technology}")

filtered_results = index.query(hybrid_query)

print(" Hybrid Search Results (Technology category only):")
print("=" * 55)
for i, result in enumerate(filtered_results, 1):
    distance = result.get('vector_distance', 'N/A')
    print(f"{i}. [{result['category']}] {result['content']}")
    print(f"    Distance: {distance:.4f}" if isinstance(distance, (int, float)) else f"    ↪Distance: {distance}")
    print()
```

```
Hybrid Search Results (Technology category only):
=====
```

Performance Monitoring

Let's examine the index statistics to understand the performance characteristics of our SVS-VAMANA index.

```
# Get detailed index information
info = index.info()

print(" Index Statistics:")
print("=" * 30)
print(f"Documents: {info.get('num_docs', 0)}")

# Handle vector_index_sz_mb which might be a string
vector_size = info.get('vector_index_sz_mb', 0)
```

(continues on next page)

(continued from previous page)

```

if isinstance(vector_size, str):
    try:
        vector_size = float(vector_size)
    except ValueError:
        vector_size = 0.0
print(f"Vector index size: {vector_size:.2f} MB")

# Handle total_indexing_time which might also be a string
indexing_time = info.get('total_indexing_time', 0)
if isinstance(indexing_time, str):
    try:
        indexing_time = float(indexing_time)
    except ValueError:
        indexing_time = 0.0
print(f"Total indexing time: {indexing_time:.2f} seconds")

# Calculate memory efficiency
if info.get('num_docs', 0) > 0 and vector_size > 0:
    mb_per_doc = vector_size / info.get('num_docs', 1)
    print(f"Memory per document: {mb_per_doc:.4f} MB")

    # Estimate for larger datasets
    for scale in [1000, 10000, 100000]:
        estimated_mb = mb_per_doc * scale
        print(f"Estimated size for {scale:,} docs: {estimated_mb:.1f} MB")
else:
    print("Memory efficiency calculation requires documents and vector index size > 0")

```

```

Index Statistics:
=====
Documents: 0
Vector index size: 0.00 MB
Total indexing time: 1.58 seconds
Memory efficiency calculation requires documents and vector index size > 0

```

Manual Configuration (Advanced)

For advanced users who want full control over SVS-VAMANA parameters, you can manually configure the algorithm instead of using CompressionAdvisor.

```

# Example of manual SVS-VAMANA configuration
manual_schema = {
    "index": {
        "name": "svs_manual",
        "prefix": "manual",
    },
    "fields": [
        {"name": "content", "type": "text"},
        {
            "name": "embedding",
            "type": "vector",

```

(continues on next page)

(continued from previous page)

```

        "attrs": {
            "dims": 768,
            "algorithm": "svs-vamana",
            "datatype": "float32",
            "distance_metric": "cosine",

            # Graph construction parameters
            "graph_max_degree": 64,          # Higher = better recall, more memory
            "construction_window_size": 300, # Higher = better quality, slower build

            # Search parameters
            "search_window_size": 40,        # Higher = better recall, slower search

            # Compression settings
            "compression": "LVQ4x4",        # Choose compression type
            "training_threshold": 10000,    # Min vectors before compression
        }
    }
]
}

print("Manual SVS-VAMANA Configuration:")
print("=" * 40)
vector_attrs = manual_schema["fields"][1]["attrs"]
for key, value in vector_attrs.items():
    if key != "dims": # Skip dims as it's obvious
        print(f"  {key}: {value}")

# Calculate memory savings for this configuration
manual_savings = CompressionAdvisor.estimate_memory_savings(
    "LVQ4x4", 768, None
)
print(f"\nEstimated memory savings: {manual_savings}%")

```

Manual SVS-VAMANA Configuration:

=====

```

algorithm: svs-vamana
datatype: float32
distance_metric: cosine
graph_max_degree: 64
construction_window_size: 300
search_window_size: 40
compression: LVQ4x4
training_threshold: 10000

```

Estimated memory savings: 75.0%

Best Practices and Tips

When to Use SVS-VAMANA

- **Large datasets** (>10K vectors) where memory efficiency matters
- **High-dimensional vectors** (>512 dimensions) that benefit from compression
- **Applications** that can tolerate slight recall trade-offs for speed and memory savings

Parameter Tuning Guidelines

- **Start with CompressionAdvisor** recommendations
- **Increase search_window_size** if you need higher recall
- **Use LeanVec** for high-dimensional vectors (1024 dims)
- **Use LVQ** for lower-dimensional vectors (<1024 dims)

Performance Considerations

- **Index build time** increases with higher construction_window_size
- **Search latency** increases with higher search_window_size
- **Memory usage** decreases with more aggressive compression
- **Recall quality** may decrease with more aggressive compression

Cleanup

Clean up the indices created in this demo.

```
# Clean up demo indices
try:
    index.delete()
    print("Cleaned up svd_demo index")
except:
    print("- svd_demo index was already deleted or doesn't exist")
```

```
Cleaned up svd_demo index
```

2.3.11 Caching Embeddings

RedisVL provides an `EmbeddingsCache` that makes it easy to store and retrieve embedding vectors with their associated text and metadata. This cache is particularly useful for applications that frequently compute the same embeddings, enabling you to:

- Reduce computational costs by reusing previously computed embeddings
- Decrease latency in applications that rely on embeddings
- Store additional metadata alongside embeddings for richer applications

This notebook will show you how to use the `EmbeddingsCache` effectively in your applications.

Setup

First, let's import the necessary libraries. We'll use a text embedding model from HuggingFace to generate our embeddings.

```
import os
import time
import numpy as np

# Disable tokenizers parallelism to avoid deadlocks
os.environ["TOKENIZERS_PARALLELISM"] = "False"

# Import the EmbeddingsCache
from redisvl.extensions.cache.embeddings import EmbeddingsCache
from redisvl.utils.vectorize import HFTextVectorizer
```

Let's create a vectorizer to generate embeddings for our texts:

```
# Initialize the vectorizer
vectorizer = HFTextVectorizer(
    model="redis/langcache-embed-v1",
    cache_folder=os.getenv("SENTENCE_TRANSFORMERS_HOME")
)
```

```
/Users/tyler.hutcherson/Documents/AppliedAI/redis-vl-python/.venv/lib/python3.13/site-
packages/tqdm/auto.py:21: TqdmWarning: IPProgress not found. Please update jupyter and
ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
from .autonotebook import tqdm as notebook_tqdm
```

```
13:06:09 sentence_transformers.SentenceTransformer INFO    Use pytorch device_name: mps
13:06:09 sentence_transformers.SentenceTransformer INFO    Load pretrained
↳ SentenceTransformer: redis/langcache-embed-v1
13:06:09 sentence_transformers.SentenceTransformer WARNING  You try to use a model that
↳ was created with version 4.1.0, however, your version is 3.4.1. This might cause
↳ unexpected behavior or errors. In that case, try to update to the latest version.
```

```
Batches: 100%|| 1/1 [00:00<00:00, 4.09it/s]
```

Initializing the EmbeddingsCache

Now let's initialize our EmbeddingsCache. The cache requires a Redis connection to store the embeddings and their associated data.

```
# Initialize the embeddings cache
cache = EmbeddingsCache(
    name="embedcache",          # name prefix for Redis keys
    redis_url="redis://localhost:6379", # Redis connection URL
    ttl=None                    # Optional TTL in seconds (None means no
↳ expiration)
)
```

Basic Usage

Storing Embeddings

Let's store some text with its embedding in the cache. The `set` method takes the following parameters:

- `text`: The input text that was embedded
- `model_name`: The name of the embedding model used
- `embedding`: The embedding vector
- `metadata`: Optional metadata associated with the embedding
- `ttl`: Optional time-to-live override for this specific entry

```
# Text to embed
text = "What is machine learning?"
model_name = "redis/langcache-embed-v1"

# Generate the embedding
embedding = vectorizer.embed(text)

# Optional metadata
metadata = {"category": "ai", "source": "user_query"}

# Store in cache
key = cache.set(
    text=text,
    model_name=model_name,
    embedding=embedding,
    metadata=metadata
)

print(f"Stored with key: {key[:15]}...")
```

```
Batches: 100%| 1/1 [00:00<00:00, 3.18it/s]
```

```
Stored with key: embedcache:909f...
```

Retrieving Embeddings

To retrieve an embedding from the cache, use the `get` method with the original text and model name:

```
# Retrieve from cache

if result := cache.get(text=text, model_name=model_name):
    print(f"Found in cache: {result['text']}")
    print(f"Model: {result['model_name']}")
    print(f"Metadata: {result['metadata']}")
    print(f"Embedding shape: {np.array(result['embedding']).shape}")
```

(continues on next page)

(continued from previous page)

```
else:
    print("Not found in cache.")
```

```
Found in cache: What is machine learning?
Model: redis/langcache-embed-v1
Metadata: {'category': 'ai', 'source': 'user_query'}
Embedding shape: (768,)
```

Checking Existence

You can check if an embedding exists in the cache without retrieving it using the `exists` method:

```
# Check if existing text is in cache
exists = cache.exists(text=text, model_name=model_name)
print(f"First query exists in cache: {exists}")

# Check if a new text is in cache
new_text = "What is deep learning?"
exists = cache.exists(text=new_text, model_name=model_name)
print(f"New query exists in cache: {exists}")
```

```
First query exists in cache: True
New query exists in cache: False
```

Removing Entries

To remove an entry from the cache, use the `drop` method:

```
# Remove from cache
cache.drop(text=text, model_name=model_name)

# Verify it's gone
exists = cache.exists(text=text, model_name=model_name)
print(f"After dropping: {exists}")
```

```
After dropping: False
```

Advanced Usage

Key-Based Operations

The `EmbeddingsCache` also provides methods that work directly with Redis keys, which can be useful for advanced use cases:

```
# Store an entry again
key = cache.set(
    text=text,
```

(continues on next page)

(continued from previous page)

```

    model_name=model_name,
    embedding=embedding,
    metadata=metadata
)
print(f"Stored with key: {key[:15]}...")

# Check existence by key
exists_by_key = cache.exists_by_key(key)
print(f"Exists by key: {exists_by_key}")

# Retrieve by key
result_by_key = cache.get_by_key(key)
print(f"Retrieved by key: {result_by_key['text']}")

# Drop by key
cache.drop_by_key(key)

```

```

Stored with key: embedcache:909f...
Exists by key: True
Retrieved by key: What is machine learning?

```

Batch Operations

When working with multiple embeddings, batch operations can significantly improve performance by reducing network roundtrips. The EmbeddingsCache provides methods prefixed with `m` (for “multi”) that handle batches efficiently.

```

# Create multiple embeddings
texts = [
    "What is machine learning?",
    "How do neural networks work?",
    "What is deep learning?"
]
embeddings = [vectorizer.embed(t) for t in texts]

# Prepare batch items as dictionaries
batch_items = [
    {
        "text": texts[0],
        "model_name": model_name,
        "embedding": embeddings[0],
        "metadata": {"category": "ai", "type": "question"}
    },
    {
        "text": texts[1],
        "model_name": model_name,
        "embedding": embeddings[1],
        "metadata": {"category": "ai", "type": "question"}
    },
    {
        "text": texts[2],

```

(continues on next page)

(continued from previous page)

```

        "model_name": model_name,
        "embedding": embeddings[2],
        "metadata": {"category": "ai", "type": "question"}
    }
]

# Store multiple embeddings in one operation
keys = cache.mset(batch_items)
print(f"Stored {len(keys)} embeddings with batch operation")

# Check if multiple embeddings exist in one operation
exist_results = cache.mexists(texts, model_name)
print(f"All embeddings exist: {all(exist_results)}")

# Retrieve multiple embeddings in one operation
results = cache.mget(texts, model_name)
print(f"Retrieved {len(results)} embeddings in one operation")

# Delete multiple embeddings in one operation
cache.mdrop(texts, model_name)

# Alternative: key-based batch operations
# cache.mget_by_keys(keys)      # Retrieve by keys
# cache.mexists_by_keys(keys)   # Check existence by keys
# cache.mdrop_by_keys(keys)     # Delete by keys

```

```

Batches: 100%| 1/1 [00:00<00:00, 21.37it/s]
Batches: 100%| 1/1 [00:00<00:00, 9.04it/s]
Batches: 100%| 1/1 [00:00<00:00, 20.84it/s]

```

```

Stored 3 embeddings with batch operation
All embeddings exist: True
Retrieved 3 embeddings in one operation

```

Batch operations are particularly beneficial when working with large numbers of embeddings. They provide the same functionality as individual operations but with better performance by reducing network roundtrips.

For asynchronous applications, async versions of all batch methods are also available with the `am` prefix (e.g., `amset`, `amexists`, `amdrop`).

Working with TTL (Time-To-Live)

You can set a global TTL when initializing the cache, or specify TTL for individual entries:

```
# Create a cache with a default 5-second TTL
ttl_cache = EmbeddingsCache(
    name="ttl_cache",
    redis_url="redis://localhost:6379",
    ttl=5 # 5 second TTL
)

# Store an entry
key = ttl_cache.set(
    text=text,
    model_name=model_name,
    embedding=embedding
)

# Check if it exists
exists = ttl_cache.exists_by_key(key)
print(f"Immediately after setting: {exists}")

# Wait for it to expire
time.sleep(6)

# Check again
exists = ttl_cache.exists_by_key(key)
print(f"After waiting: {exists}")
```

```
Immediately after setting: True
After waiting: False
```

You can also override the default TTL for individual entries:

```
# Store an entry with a custom 1-second TTL
key1 = ttl_cache.set(
    text="Short-lived entry",
    model_name=model_name,
    embedding=embedding,
    ttl=1 # Override with 1 second TTL
)

# Store another entry with the default TTL (5 seconds)
key2 = ttl_cache.set(
    text="Default TTL entry",
    model_name=model_name,
    embedding=embedding
    # No TTL specified = uses the default 5 seconds
)

# Wait for 2 seconds
time.sleep(2)
```

(continues on next page)

(continued from previous page)

```

# Check both entries
exists1 = ttl_cache.exists_by_key(key1)
exists2 = ttl_cache.exists_by_key(key2)

print(f"Entry with custom TTL after 2 seconds: {exists1}")
print(f"Entry with default TTL after 2 seconds: {exists2}")

# Cleanup
ttl_cache.drop_by_key(key2)

```

```

Entry with custom TTL after 2 seconds: False
Entry with default TTL after 2 seconds: True

```

Async Support

The `EmbeddingsCache` provides async versions of all methods for use in async applications. The async methods are prefixed with a (e.g., `aset`, `aget`, `aexists`, `adrop`).

```

async def async_cache_demo():
    # Store an entry asynchronously
    key = await cache.aset(
        text="Async embedding",
        model_name=model_name,
        embedding=embedding,
        metadata={"async": True}
    )

    # Check if it exists
    exists = await cache.aexists_by_key(key)
    print(f"Async set successful? {exists}")

    # Retrieve it
    result = await cache.aget_by_key(key)
    success = result is not None and result["text"] == "Async embedding"
    print(f"Async get successful? {success}")

    # Remove it
    await cache.adrop_by_key(key)

# Run the async demo
await async_cache_demo()

```

```

Async set successful? True
Async get successful? True

```

Real-World Example

Let's build a simple embeddings caching system for a text classification task. We'll check the cache before computing new embeddings to save computation time.

```
# Create a fresh cache for this example
example_cache = EmbeddingsCache(
    name="example_cache",
    redis_url="redis://localhost:6379",
    ttl=3600 # 1 hour TTL
)

vectorizer = HFTextVectorizer(
    model=model_name,
    cache=example_cache,
    cache_folder=os.getenv("SENTENCE_TRANSFORMERS_HOME")
)

# Simulate processing a stream of queries
queries = [
    "What is artificial intelligence?",
    "How does machine learning work?",
    "What is artificial intelligence?", # Repeated query
    "What are neural networks?",
    "How does machine learning work?" # Repeated query
]

# Process the queries and track statistics
total_queries = 0
cache_hits = 0

for query in queries:
    total_queries += 1

    # Check cache before computing
    before = example_cache.exists(text=query, model_name=model_name)
    if before:
        cache_hits += 1

    # Get embedding (will compute or use cache)
    embedding = vectorizer.embed(query)

# Report statistics
cache_misses = total_queries - cache_hits
hit_rate = (cache_hits / total_queries) * 100

print("\nStatistics:")
print(f"Total queries: {total_queries}")
print(f"Cache hits: {cache_hits}")
print(f"Cache misses: {cache_misses}")
print(f"Cache hit rate: {hit_rate:.1f}%")

# Cleanup
```

(continues on next page)

(continued from previous page)

```
for query in set(queries): # Use set to get unique queries
    example_cache.drop(text=query, model_name=model_name)
```

```
13:06:20 sentence_transformers.SentenceTransformer INFO    Use pytorch device_name: mps
13:06:20 sentence_transformers.SentenceTransformer INFO    Load pretrained_
↳ SentenceTransformer: redis/langcache-embed-v1
13:06:20 sentence_transformers.SentenceTransformer WARNING  You try to use a model that_
↳ was created with version 4.1.0, however, your version is 3.4.1. This might cause_
↳ unexpected behavior or errors. In that case, try to update to the latest version.
```

```
Batches: 100%| 1/1 [00:00<00:00, 21.84it/s]
Batches: 100%| 1/1 [00:00<00:00, 22.04it/s]
Batches: 100%| 1/1 [00:00<00:00, 22.62it/s]
Batches: 100%| 1/1 [00:00<00:00, 22.71it/s]
```

```
Statistics:
Total queries: 5
Cache hits: 2
Cache misses: 3
Cache hit rate: 40.0%
```

Performance Benchmark

Let's run benchmarks to compare the performance of embedding with and without caching, as well as batch versus individual operations.

```
# Text to use for benchmarking
benchmark_text = "This is a benchmark text to measure the performance of embedding_
↳ caching."

# Create a fresh cache for benchmarking
benchmark_cache = EmbeddingsCache(
    name="benchmark_cache",
    redis_url="redis://localhost:6379",
    ttl=3600 # 1 hour TTL
)
vectorizer.cache = benchmark_cache

# Number of iterations for the benchmark
n_iterations = 10

# Benchmark without caching
print("Benchmarking without caching:")
start_time = time.time()
for _ in range(n_iterations):
    embedding = vectorizer.embed(text, skip_cache=True)
no_cache_time = time.time() - start_time
print(f"Time taken without caching: {no_cache_time:.4f} seconds")
```

(continues on next page)

(continued from previous page)

```

print(f"Average time per embedding: {no_cache_time/n_iterations:.4f} seconds")

# Benchmark with caching
print("\nBenchmarking with caching:")
start_time = time.time()
for _ in range(n_iterations):
    embedding = vectorizer.embed(text)
cache_time = time.time() - start_time
print(f"Time taken with caching: {cache_time:.4f} seconds")
print(f"Average time per embedding: {cache_time/n_iterations:.4f} seconds")

# Compare performance
speedup = no_cache_time / cache_time
latency_reduction = (no_cache_time/n_iterations) - (cache_time/n_iterations)
print(f"\nPerformance comparison:")
print(f"Speedup with caching: {speedup:.2f}x faster")
print(f"Time saved: {no_cache_time - cache_time:.4f} seconds ({(1 - cache_time/no_cache_
↪time) * 100:.1f}%)")
print(f"Latency reduction: {latency_reduction:.4f} seconds per query")

```

Benchmarking without caching:

```

Batches: 100%| 1/1 [00:00<00:00, 21.51it/s]
Batches: 100%| 1/1 [00:00<00:00, 23.21it/s]
Batches: 100%| 1/1 [00:00<00:00, 23.96it/s]
Batches: 100%| 1/1 [00:00<00:00, 23.28it/s]
Batches: 100%| 1/1 [00:00<00:00, 22.69it/s]
Batches: 100%| 1/1 [00:00<00:00, 22.98it/s]
Batches: 100%| 1/1 [00:00<00:00, 23.17it/s]
Batches: 100%| 1/1 [00:00<00:00, 24.12it/s]
Batches: 100%| 1/1 [00:00<00:00, 23.37it/s]
Batches: 100%| 1/1 [00:00<00:00, 23.24it/s]

```

Time taken without caching: 0.4549 seconds
Average time per embedding: 0.0455 seconds

Benchmarking with caching:

```

Batches: 100%| 1/1 [00:00<00:00, 23.69it/s]

```

Time taken with caching: 0.0664 seconds
Average time per embedding: 0.0066 seconds

Performance comparison:
Speedup with caching: 6.86x faster
Time saved: 0.3885 seconds (85.4%)
Latency reduction: 0.0389 seconds per query

Common Use Cases for Embedding Caching

Embedding caching is particularly useful in the following scenarios:

1. **Search applications:** Cache embeddings for frequently searched queries to reduce latency
2. **Content recommendation systems:** Cache embeddings for content items to speed up similarity calculations
3. **API services:** Reduce costs and improve response times when generating embeddings through paid APIs
4. **Batch processing:** Speed up processing of datasets that contain duplicate texts
5. **Chatbots and virtual assistants:** Cache embeddings for common user queries to provide faster responses
6. **Development workflows**

Cleanup

Let's clean up our caches to avoid leaving data in Redis:

```
# Clean up all caches
cache.clear()
ttl_cache.clear()
example_cache.clear()
benchmark_cache.clear()
```

Summary

The `EmbeddingsCache` provides an efficient way to store and retrieve embeddings with their associated text and meta-data. Key features include:

- Simple API for storing and retrieving individual embeddings (`set/get`)
- Batch operations for working with multiple embeddings efficiently (`mset/mget/mexists/mdrop`)
- Support for metadata storage alongside embeddings
- Configurable time-to-live (TTL) for cache entries
- Key-based operations for advanced use cases
- Async support for use in asynchronous applications
- Significant performance improvements (15-20x faster with batch operations)

By using the `EmbeddingsCache`, you can reduce computational costs and improve the performance of applications that rely on embeddings.

2.3.12 Advanced Query Types

In this notebook, we will explore advanced query types available in RedisVL:

1. **TextQuery:** Full text search with advanced scoring
2. **AggregateHybridQuery:** Combines text and vector search for hybrid retrieval
3. **MultiVectorQuery:** Search over multiple vector fields simultaneously

These query types are powerful tools for building sophisticated search applications that go beyond simple vector similarity search.

Prerequisites:

- Ensure `redisvl` is installed in your Python environment.
- Have a running instance of [Redis Stack](#) or [Redis Cloud](#).

Setup and Data Preparation

First, let's create a schema and prepare sample data that includes text fields, numeric fields, and vector fields.

```
import numpy as np
from jupyterutils import result_print

# Sample data with text descriptions, categories, and vectors
data = [
    {
        'product_id': 'prod_1',
        'brief_description': 'comfortable running shoes for athletes',
        'full_description': 'Engineered with a dual-layer EVA foam midsole and FlexWeave_
↪breathable mesh upper, these running shoes deliver responsive cushioning for long-
↪distance runs. The anatomical footbed adapts to your stride while the carbon rubber_
↪outsole provides superior traction on varied terrain.',
        'category': 'footwear',
        'price': 89.99,
        'rating': 4.5,
        'text_embedding': np.array([0.1, 0.2, 0.1], dtype=np.float32).tobytes(),
        'image_embedding': np.array([0.8, 0.1], dtype=np.float32).tobytes(),
    },
    {
        'product_id': 'prod_2',
        'brief_description': 'lightweight running jacket with water resistance',
        'full_description': 'Stay protected with this ultralight 2.5-layer DWR-coated_
↪shell featuring laser-cut ventilation zones and reflective piping for low-light_
↪visibility. Packs into its own chest pocket and weighs just 4.2 oz, making it ideal_
↪for unpredictable weather conditions.',
        'category': 'outerwear',
        'price': 129.99,
        'rating': 4.8,
        'text_embedding': np.array([0.2, 0.3, 0.2], dtype=np.float32).tobytes(),
        'image_embedding': np.array([0.7, 0.2], dtype=np.float32).tobytes(),
    },
    {
        'product_id': 'prod_3',
        'brief_description': 'professional tennis racket for competitive players',
        'full_description': 'Competition-grade racket featuring a 98 sq in head size,_
↪16x19 string pattern, and aerospace-grade graphite frame that delivers explosive power_
↪with pinpoint control. Tournament-approved specs include 315g weight and 68 RA_
↪stiffness rating for advanced baseline play.',
        'category': 'equipment',
        'price': 199.99,
        'rating': 4.9,
```

(continues on next page)

(continued from previous page)

```

        'text_embedding': np.array([0.9, 0.1, 0.05], dtype=np.float32).tobytes(),
        'image_embedding': np.array([0.1, 0.9], dtype=np.float32).tobytes(),
    },
    {
        'product_id': 'prod_4',
        'brief_description': 'yoga mat with extra cushioning for comfort',
        'full_description': 'Premium 8mm thick TPE yoga mat with dual-texture surface -
        ↪smooth side for hot yoga flow and textured side for maximum grip during balancing
        ↪poses. Closed-cell technology prevents moisture absorption while alignment markers
        ↪guide proper positioning in asanas.',
        'category': 'accessories',
        'price': 39.99,
        'rating': 4.3,
        'text_embedding': np.array([0.15, 0.25, 0.15], dtype=np.float32).tobytes(),
        'image_embedding': np.array([0.5, 0.5], dtype=np.float32).tobytes(),
    },
    {
        'product_id': 'prod_5',
        'brief_description': 'basketball shoes with excellent ankle support',
        'full_description': 'High-top basketball sneakers with Zoom Air units in
        ↪forefoot and heel, reinforced lateral sidewalls for explosive cuts, and herringbone
        ↪traction pattern optimized for hardwood courts. The internal bootie construction and
        ↪extended ankle collar provide lockdown support during aggressive drives.',
        'category': 'footwear',
        'price': 139.99,
        'rating': 4.7,
        'text_embedding': np.array([0.12, 0.18, 0.12], dtype=np.float32).tobytes(),
        'image_embedding': np.array([0.75, 0.15], dtype=np.float32).tobytes(),
    },
    {
        'product_id': 'prod_6',
        'brief_description': 'swimming goggles with anti-fog coating',
        'full_description': 'Low-profile competition goggles with curved polycarbonate
        ↪lenses offering 180-degree peripheral vision and UV protection. Hydrophobic anti-fog
        ↪coating lasts 10x longer than standard treatments, while the split silicone strap and
        ↪interchangeable nose bridges ensure a watertight, custom fit.',
        'category': 'accessories',
        'price': 24.99,
        'rating': 4.4,
        'text_embedding': np.array([0.3, 0.1, 0.2], dtype=np.float32).tobytes(),
        'image_embedding': np.array([0.2, 0.8], dtype=np.float32).tobytes(),
    },
]

```

Define the Schema

Our schema includes:

- **Tag fields:** `product_id`, `category`
- **Text fields:** `brief_description` and `full_description` for full-text search
- **Numeric fields:** `price`, `rating`
- **Vector fields:** `text_embedding` (3 dimensions) and `image_embedding` (2 dimensions) for semantic search

```
schema = {
  "index": {
    "name": "advanced_queries",
    "prefix": "products",
    "storage_type": "hash",
  },
  "fields": [
    {"name": "product_id", "type": "tag"},
    {"name": "category", "type": "tag"},
    {"name": "brief_description", "type": "text"},
    {"name": "full_description", "type": "text"},
    {"name": "price", "type": "numeric"},
    {"name": "rating", "type": "numeric"},
    {
      "name": "text_embedding",
      "type": "vector",
      "attrs": {
        "dims": 3,
        "distance_metric": "cosine",
        "algorithm": "flat",
        "datatype": "float32"
      }
    },
    {
      "name": "image_embedding",
      "type": "vector",
      "attrs": {
        "dims": 2,
        "distance_metric": "cosine",
        "algorithm": "flat",
        "datatype": "float32"
      }
    }
  ],
}
```

Create Index and Load Data

```
from redisvl.index import SearchIndex

# Create the search index
index = SearchIndex.from_dict(schema, redis_url="redis://localhost:6379")

# Create the index and load data
index.create(overwrite=True)
keys = index.load(data)

print(f"Loaded {len(keys)} products into the index")
```

```
Loaded 6 products into the index
```

1. TextQuery: Full Text Search

The `TextQuery` class enables full text search with advanced scoring algorithms. It's ideal for keyword-based search with relevance ranking.

Basic Text Search

Let's search for products related to "running shoes":

```
from redisvl.query import TextQuery

# Create a text query
text_query = TextQuery(
    text="running shoes",
    text_field_name="brief_description",
    return_fields=["product_id", "brief_description", "category", "price"],
    num_results=5
)

results = index.query(text_query)
result_print(results)
```

```
<IPython.core.display.HTML object>
```

Text Search with Different Scoring Algorithms

RedisVL supports multiple text scoring algorithms. Let's compare BM25STD and TFIDF:

```
# BM25 standard scoring (default)
bm25_query = TextQuery(
    text="comfortable shoes",
    text_field_name="brief_description",
    text_scorer="BM25STD",
    return_fields=["product_id", "brief_description", "price"],
```

(continues on next page)

(continued from previous page)

```

    num_results=3
)

print("Results with BM25 scoring:")
results = index.query(bm25_query)
result_print(results)

```

Results with BM25 scoring:

<IPython.core.display.HTML object>

```

# TFIDF scoring
tfidf_query = TextQuery(
    text="comfortable shoes",
    text_field_name="brief_description",
    text_scorer="TFIDF",
    return_fields=["product_id", "brief_description", "price"],
    num_results=3
)

print("Results with TFIDF scoring:")
results = index.query(tfidf_query)
result_print(results)

```

Results with TFIDF scoring:

<IPython.core.display.HTML object>

Text Search with Filters

Combine text search with filters to narrow results:

```

from redisvl.query.filter import Tag, Num

# Search for "shoes" only in the footwear category
filtered_text_query = TextQuery(
    text="shoes",
    text_field_name="brief_description",
    filter_expression=Tag("category") == "footwear",
    return_fields=["product_id", "brief_description", "category", "price"],
    num_results=5
)

results = index.query(filtered_text_query)
result_print(results)

```

<IPython.core.display.HTML object>

```
# Search for products under $100
price_filtered_query = TextQuery(
    text="comfortable",
    text_field_name="brief_description",
    filter_expression=Num("price") < 100,
    return_fields=["product_id", "brief_description", "price"],
    num_results=5
)

results = index.query(price_filtered_query)
result_print(results)
```

```
<IPython.core.display.HTML object>
```

Text Search with Multiple Fields and Weights

You can search across multiple text fields with different weights to prioritize certain fields. Here we'll prioritize the `brief_description` field and make text similarity in that field twice as important as text similarity in `full_description`:

```
weighted_query = TextQuery(
    text="shoes",
    text_field_name={"brief_description": 1.0, "full_description": 0.5},
    return_fields=["product_id", "brief_description"],
    num_results=3
)

results = index.query(weighted_query)
result_print(results)
```

```
<IPython.core.display.HTML object>
```

Text Search with Custom Stopwords

Stopwords are common words that are filtered out before processing the query. You can specify which language's default stopwords should be filtered out, like `english`, `french`, or `german`. You can also define your own list of stopwords:

```
# Use English stopwords (default)
query_with_stopwords = TextQuery(
    text="the best shoes for running",
    text_field_name="brief_description",
    stopwords="english", # Common words like "the", "for" will be removed
    return_fields=["product_id", "brief_description"],
    num_results=3
)

results = index.query(query_with_stopwords)
result_print(results)
```

```
<IPython.core.display.HTML object>
```

```
# Use custom stopwords
custom_stopwords_query = TextQuery(
    text="professional equipment for athletes",
    text_field_name="brief_description",
    stopwords=["for", "with"], # Only these words will be filtered
    return_fields=["product_id", "brief_description"],
    num_results=3
)

results = index.query(custom_stopwords_query)
result_print(results)
```

```
<IPython.core.display.HTML object>
```

```
# No stopwords
no_stopwords_query = TextQuery(
    text="the best shoes for running",
    text_field_name="brief_description",
    stopwords=None, # All words will be included
    return_fields=["product_id", "brief_description"],
    num_results=3
)

results = index.query(no_stopwords_query)
result_print(results)
```

```
<IPython.core.display.HTML object>
```

2. AggregateHybridQuery: Combining Text and Vector Search

The `AggregateHybridQuery` combines text search and vector similarity to provide the best of both worlds:

- **Text search:** Finds exact keyword matches
- **Vector search:** Captures semantic similarity

Results are scored using a weighted combination:

```
hybrid_score = (alpha) * vector_score + (1 - alpha) * text_score
```

Where `alpha` controls the balance between vector and text search (default: 0.7).

Basic Aggregate Hybrid Query

Let's search for “running” with both text and semantic search:

```
from redisvl.query import AggregateHybridQuery

# Create a hybrid query
hybrid_query = AggregateHybridQuery(
    text="running shoes",
    text_field_name="brief_description",
    vector=[0.1, 0.2, 0.1], # Query vector
    vector_field_name="text_embedding",
    return_fields=["product_id", "brief_description", "category", "price"],
    num_results=5
)

results = index.query(hybrid_query)
result_print(results)
```

```
<IPython.core.display.HTML object>
```

Adjusting the Alpha Parameter

The alpha parameter controls the weight between vector and text search:

- `alpha=1.0`: Pure vector search
- `alpha=0.0`: Pure text search
- `alpha=0.7` (default): 70% vector, 30% text

```
# More emphasis on vector search (alpha=0.9)
vector_heavy_query = AggregateHybridQuery(
    text="comfortable",
    text_field_name="brief_description",
    vector=[0.15, 0.25, 0.15],
    vector_field_name="text_embedding",
    alpha=0.9, # 90% vector, 10% text
    return_fields=["product_id", "brief_description"],
    num_results=3
)

print("Results with alpha=0.9 (vector-heavy):")
results = index.query(vector_heavy_query)
result_print(results)
```

```
Results with alpha=0.9 (vector-heavy):
```

```
<IPython.core.display.HTML object>
```

Aggregate Hybrid Query with Filters

You can also combine hybrid search with filters:

```
# Hybrid search with a price filter
filtered_hybrid_query = AggregateHybridQuery(
    text="professional equipment",
    text_field_name="brief_description",
    vector=[0.9, 0.1, 0.05],
    vector_field_name="text_embedding",
    filter_expression=Num("price") > 100,
    return_fields=["product_id", "brief_description", "category", "price"],
    num_results=5
)

results = index.query(filtered_hybrid_query)
result_print(results)
```

<IPython.core.display.HTML object>

Using Different Text Scorers

AggregateHybridQuery supports the same text scoring algorithms as TextQuery:

```
# Aggregate Hybrid query with TFIDF scorer
hybrid_tfidf = AggregateHybridQuery(
    text="shoes support",
    text_field_name="brief_description",
    vector=[0.12, 0.18, 0.12],
    vector_field_name="text_embedding",
    text_scorer="TFIDF",
    return_fields=["product_id", "brief_description"],
    num_results=3
)

results = index.query(hybrid_tfidf)
result_print(results)
```

<IPython.core.display.HTML object>

3. MultiVectorQuery: Multi-Vector Search

The MultiVectorQuery allows you to search over multiple vector fields simultaneously. This is useful when you have different types of embeddings (e.g., text and image embeddings) and want to find results that match across multiple modalities.

The final score is calculated as a weighted combination:

```
combined_score = w_1 * score_1 + w_2 * score_2 + w_3 * score_3 + ...
```

Basic Multi-Vector Query

First, we need to import the `Vector` class to define our query vectors:

```
from redisvl.query import MultiVectorQuery, Vector

# Define multiple vectors for the query
text_vector = Vector(
    vector=[0.1, 0.2, 0.1],
    field_name="text_embedding",
    dtype="float32",
    weight=0.7 # 70% weight for text embedding
)

image_vector = Vector(
    vector=[0.8, 0.1],
    field_name="image_embedding",
    dtype="float32",
    weight=0.3 # 30% weight for image embedding
)

# Create a multi-vector query
multi_vector_query = MultiVectorQuery(
    vectors=[text_vector, image_vector],
    return_fields=["product_id", "brief_description", "category"],
    num_results=5
)

results = index.query(multi_vector_query)
result_print(results)
```

```
<IPython.core.display.HTML object>
```

Adjusting Vector Weights

You can adjust the weights to prioritize different vector fields:

```
# More emphasis on image similarity
text_vec = Vector(
    vector=[0.9, 0.1, 0.05],
    field_name="text_embedding",
    dtype="float32",
    weight=0.2 # 20% weight
)

image_vec = Vector(
    vector=[0.1, 0.9],
    field_name="image_embedding",
    dtype="float32",
    weight=0.8 # 80% weight
)
```

(continues on next page)

(continued from previous page)

```

image_heavy_query = MultiVectorQuery(
    vectors=[text_vec, image_vec],
    return_fields=["product_id", "brief_description", "category"],
    num_results=3
)

print("Results with emphasis on image similarity:")
results = index.query(image_heavy_query)
result_print(results)

```

Results with emphasis on image similarity:

<IPython.core.display.HTML object>

Multi-Vector Query with Filters

Combine multi-vector search with filters to narrow results:

```

# Multi-vector search with category filter
text_vec = Vector(
    vector=[0.1, 0.2, 0.1],
    field_name="text_embedding",
    dtype="float32",
    weight=0.6
)

image_vec = Vector(
    vector=[0.8, 0.1],
    field_name="image_embedding",
    dtype="float32",
    weight=0.4
)

filtered_multi_query = MultiVectorQuery(
    vectors=[text_vec, image_vec],
    filter_expression=Tag("category") == "footwear",
    return_fields=["product_id", "brief_description", "category", "price"],
    num_results=5
)

results = index.query(filtered_multi_query)
result_print(results)

```

<IPython.core.display.HTML object>

Comparing Query Types

Let's compare the three query types side by side:

```
# TextQuery - keyword-based search
text_q = TextQuery(
    text="shoes",
    text_field_name="brief_description",
    return_fields=["product_id", "brief_description"],
    num_results=3
)

print("TextQuery Results (keyword-based):")
result_print(index.query(text_q))
print()
```

TextQuery Results (keyword-based):

<IPython.core.display.HTML object>

```
# AggregateHybridQuery - combines text and vector search
hybrid_q = AggregateHybridQuery(
    text="shoes",
    text_field_name="brief_description",
    vector=[0.1, 0.2, 0.1],
    vector_field_name="text_embedding",
    return_fields=["product_id", "brief_description"],
    num_results=3
)

print("AggregateHybridQuery Results (text + vector):")
result_print(index.query(hybrid_q))
print()
```

AggregateHybridQuery Results (text + vector):

<IPython.core.display.HTML object>

```
# MultiVectorQuery - searches multiple vector fields
mv_text = Vector(
    vector=[0.1, 0.2, 0.1],
    field_name="text_embedding",
    dtype="float32",
    weight=0.5
)

mv_image = Vector(
```

(continues on next page)

(continued from previous page)

```

    vector=[0.8, 0.1],
    field_name="image_embedding",
    dtype="float32",
    weight=0.5
)

multi_q = MultiVectorQuery(
    vectors=[mv_text, mv_image],
    return_fields=["product_id", "brief_description"],
    num_results=3
)

print("MultiVectorQuery Results (multiple vectors):")
result_print(index.query(multi_q))

```

```
MultiVectorQuery Results (multiple vectors):
```

```
<IPython.core.display.HTML object>
```

Best Practices

When to Use Each Query Type:

1. **TextQuery:**

- When you need precise keyword matching
- For traditional search engine functionality
- When text relevance scoring is important
- Example: Product search, document retrieval

2. **AggregateHybridQuery:**

- When you want to combine keyword and semantic search
- For improved search quality over pure text or vector search
- When you have both text and vector representations of your data
- Example: E-commerce search, content recommendation

3. **MultiVectorQuery:**

- When you have multiple types of embeddings (text, image, audio, etc.)
- For multi-modal search applications
- When you want to balance multiple semantic signals
- Example: Image-text search, cross-modal retrieval

```

# Cleanup
index.delete()

```

2.4 Example Gallery

Explore community examples of RedisVL in the wild.

 **Note**

If you are using RedisVL, please consider adding your example to this page by opening a Pull Request on [GitHub](#)

Arxiv Paper Search
Redis RAG Workbench

eCommerce Search
LLM Recommender for Hotels

Real-Time Embeddings with Redis and Bytewax
Agentic RAG

Symbols

__eq__() (Geo method), 91
 __eq__() (Num method), 89
 __eq__() (Tag method), 88
 __eq__() (Text method), 88
 __ge__() (Num method), 90
 __gt__() (Num method), 90
 __init__() (GeoRadius method), 92
 __le__() (Num method), 90
 __lt__() (Num method), 90
 __mod__() (Text method), 89
 __ne__() (Geo method), 91
 __ne__() (Num method), 90
 __ne__() (Tag method), 88
 __ne__() (Text method), 89
 __str__() (Geo method), 92
 __str__() (Num method), 91
 __str__() (Tag method), 88
 __str__() (Text method), 89

A

acheck() (SemanticCache method), 110
 aclear() (EmbeddingsCache method), 117
 aclear() (SemanticCache method), 111
 add_field() (IndexSchema method), 13
 add_fields() (IndexSchema method), 13
 add_message() (MessageHistory method), 132
 add_message() (SemanticMessageHistory method), 129
 add_messages() (MessageHistory method), 132
 add_messages() (SemanticMessageHistory method), 129
 add_route_references() (SemanticRouter method), 134
 add_scores() (HybridQuery method), 66
 add_scores() (MultiVectorQuery method), 84
 adelete() (SemanticCache method), 111
 adisconnect() (EmbeddingsCache method), 118
 adisconnect() (SemanticCache method), 111
 adrop() (EmbeddingsCache method), 118
 adrop() (SemanticCache method), 111
 adrop_by_key() (EmbeddingsCache method), 118
 aexists() (EmbeddingsCache method), 118

aexists_by_key() (EmbeddingsCache method), 118
 aexpire() (EmbeddingsCache method), 119
 aexpire() (SemanticCache method), 112
 aget() (EmbeddingsCache method), 119
 aget_by_key() (EmbeddingsCache method), 119
 aggregate() (AsyncSearchIndex method), 45
 aggregate() (SearchIndex method), 38
 aindex (SemanticCache property), 116
 algorithm (BaseVectorFieldAttributes attribute), 22
 algorithm (FlatVectorFieldAttributes attribute), 31
 algorithm (HNSWVectorFieldAttributes attribute), 24
 algorithm (SVSConfig attribute), 34
 algorithm (SVSVectorFieldAttributes attribute), 28
 amdrops() (EmbeddingsCache method), 120
 amdrops_by_keys() (EmbeddingsCache method), 120
 amexists() (EmbeddingsCache method), 120
 amexists_by_keys() (EmbeddingsCache method), 121
 amget() (EmbeddingsCache method), 121
 amget_by_keys() (EmbeddingsCache method), 121
 amset() (EmbeddingsCache method), 122
 apply() (HybridQuery method), 66
 apply() (MultiVectorQuery method), 84
 arank() (CohereReranker method), 105
 arank() (HFCrossEncoderReranker method), 107
 arank() (VoyageAIReranker method), 109
 as_redis_field() (FlatVectorField method), 30
 as_redis_field() (GeoField method), 20
 as_redis_field() (HNSWVectorField method), 23
 as_redis_field() (NumericField method), 19
 as_redis_field() (SVSVectorField method), 27
 as_redis_field() (TagField method), 18
 as_redis_field() (TextField method), 16
 aset() (EmbeddingsCache method), 122
 astore() (SemanticCache method), 112
 AsyncSearchIndex (class in redisvl.index), 45
 attrs (FlatVectorField attribute), 30
 attrs (GeoField attribute), 21
 attrs (HNSWVectorField attribute), 23
 attrs (NumericField attribute), 19
 attrs (SVSVectorField attribute), 27
 attrs (TagField attribute), 18
 attrs (TextField attribute), 17

update() (*SemanticCache* method), 113
 avg (*DistanceAggregationMethod* attribute), 140
 AzureOpenAITextVectorizer (class in *re-disvl.utils.vectorize.text.azureopenai*), 95

B

BaseVectorFieldAttributes (class in *re-disvl.schema.fields*), 21
 batch_query() (*AsyncSearchIndex* method), 45
 batch_query() (*SearchIndex* method), 38
 batch_search() (*AsyncSearchIndex* method), 46
 batch_search() (*SearchIndex* method), 38
 batch_size (*VectorQuery* property), 58
 batch_size (*VectorRangeQuery* property), 64
 BedrockTextVectorizer (class in *re-disvl.utils.vectorize.text.bedrock*), 100
 between() (*Num* method), 91
 block_size (*FlatVectorFieldAttributes* attribute), 31

C

case_sensitive (*TagFieldAttributes* attribute), 19
 check() (*SemanticCache* method), 113
 clear() (*AsyncSearchIndex* method), 46
 clear() (*EmbeddingsCache* method), 123
 clear() (*MessageHistory* method), 132
 clear() (*SearchIndex* method), 39
 clear() (*SemanticCache* method), 114
 clear() (*SemanticMessageHistory* method), 130
 clear() (*SemanticRouter* method), 134
 client (*AsyncSearchIndex* property), 52
 client (*SearchIndex* property), 44
 CohereReranker (class in *redisvl.utils.rerank.cohere*), 105
 CohereTextVectorizer (class in *re-disvl.utils.vectorize.text.cohere*), 99
 compression (*SVSConfig* attribute), 34
 compression (*SVSVectorFieldAttributes* attribute), 28
 CompressionAdvisor (class in *re-disvl.utils.compression*), 32
 connect() (*AsyncSearchIndex* method), 46
 connect() (*SearchIndex* method), 39
 construction_window_size (*SVSConfig* attribute), 35
 construction_window_size (*SVSVectorFieldAttributes* attribute), 28
 CountQuery (class in *redisvl.query*), 79
 create() (*AsyncSearchIndex* method), 46
 create() (*SearchIndex* method), 39
 CustomTextVectorizer (class in *re-disvl.utils.vectorize.text.custom*), 102

D

datatype (*BaseVectorFieldAttributes* attribute), 22
 datatype (*SVSConfig* attribute), 34
 delete() (*AsyncSearchIndex* method), 47

delete() (*MessageHistory* method), 132
 delete() (*SearchIndex* method), 39
 delete() (*SemanticCache* method), 114
 delete() (*SemanticMessageHistory* method), 130
 delete() (*SemanticRouter* method), 134
 delete_route_references() (*SemanticRouter* method), 135
 dialect() (*CountQuery* method), 79
 dialect() (*FilterQuery* method), 75
 dialect() (*HybridQuery* method), 66
 dialect() (*MultiVectorQuery* method), 84
 dialect() (*TextQuery* method), 70
 dialect() (*VectorQuery* method), 54
 dialect() (*VectorRangeQuery* method), 60
 dims (*BaseVectorFieldAttributes* attribute), 22
 disconnect() (*AsyncSearchIndex* method), 47
 disconnect() (*EmbeddingsCache* method), 123
 disconnect() (*SearchIndex* method), 40
 disconnect() (*SemanticCache* method), 114
 distance (*RouteMatch* attribute), 140
 distance_metric (*BaseVectorFieldAttributes* attribute), 22
 distance_threshold (*Route* attribute), 139
 distance_threshold (*SemanticCache* property), 116
 distance_threshold (*VectorRangeQuery* property), 64
 DistanceAggregationMethod (class in *re-disvl.extensions.router.schema*), 140
 drop() (*EmbeddingsCache* method), 123
 drop() (*MessageHistory* method), 133
 drop() (*SemanticCache* method), 114
 drop() (*SemanticMessageHistory* method), 130
 drop_by_key() (*EmbeddingsCache* method), 123
 drop_documents() (*AsyncSearchIndex* method), 47
 drop_documents() (*SearchIndex* method), 40
 drop_keys() (*AsyncSearchIndex* method), 47
 drop_keys() (*SearchIndex* method), 40

E

ef_construction (*HNSWVectorFieldAttributes* attribute), 24
 ef_runtime (*HNSWVectorFieldAttributes* attribute), 24
 ef_runtime (*VectorQuery* property), 58
 EmbeddingsCache (class in *re-disvl.extensions.cache.embeddings*), 117
 epsilon (*HNSWVectorFieldAttributes* attribute), 25
 epsilon (*SVSVectorFieldAttributes* attribute), 28
 epsilon (*VectorRangeQuery* property), 64
 estimate_memory_savings() (*CompressionAdvisor* static method), 32
 exists() (*AsyncSearchIndex* method), 48
 exists() (*EmbeddingsCache* method), 124
 exists() (*SearchIndex* method), 40
 exists_by_key() (*EmbeddingsCache* method), 124
 expander() (*CountQuery* method), 79

expander() (*FilterQuery* method), 75
 expander() (*TextQuery* method), 70
 expander() (*VectorQuery* method), 54
 expander() (*VectorRangeQuery* method), 60
 expire() (*EmbeddingsCache* method), 124
 expire() (*SemanticCache* method), 115
 expire_keys() (*AsyncSearchIndex* method), 48
 expire_keys() (*SearchIndex* method), 40

F

fetch() (*AsyncSearchIndex* method), 48
 fetch() (*SearchIndex* method), 40
 field_data (*BaseVectorFieldAttributes* property), 22
 field_names (*IndexSchema* property), 15
 field_weights (*TextQuery* property), 74
 fields (*IndexSchema* attribute), 15
 filter (*CountQuery* property), 82
 filter (*FilterQuery* property), 78
 filter (*TextQuery* property), 74
 filter (*VectorQuery* property), 58
 filter (*VectorRangeQuery* property), 64
 filter() (*HybridQuery* method), 66
 filter() (*MultiVectorQuery* method), 84
 FilterExpression (*class in redisvl.query.filter*), 87
 FilterQuery (*class in redisvl.query*), 74
 FlatVectorField (*class in redisvl.schema.fields*), 30
 FlatVectorFieldAttributes (*class in redisvl.schema.fields*), 31
 from_dict() (*AsyncSearchIndex* class method), 48
 from_dict() (*IndexSchema* class method), 14
 from_dict() (*SearchIndex* class method), 41
 from_dict() (*SemanticRouter* class method), 135
 from_existing() (*AsyncSearchIndex* class method), 48
 from_existing() (*SearchIndex* class method), 41
 from_existing() (*SemanticRouter* class method), 135
 from_yaml() (*AsyncSearchIndex* class method), 49
 from_yaml() (*IndexSchema* class method), 14
 from_yaml() (*SearchIndex* class method), 41
 from_yaml() (*SemanticRouter* class method), 135

G

Geo (*class in redisvl.query.filter*), 91
 GeoField (*class in redisvl.schema.fields*), 20
 GeoFieldAttributes (*class in redisvl.schema.fields*), 21
 GeoRadius (*class in redisvl.query.filter*), 92
 get() (*EmbeddingsCache* method), 124
 get() (*SemanticRouter* method), 136
 get_by_key() (*EmbeddingsCache* method), 125
 get_recent() (*MessageHistory* method), 133
 get_recent() (*SemanticMessageHistory* method), 130
 get_relevant() (*SemanticMessageHistory* method), 130

get_route_references() (*SemanticRouter* method), 136
 graph_max_degree (*SVSConfig* attribute), 35
 graph_max_degree (*SVSVectorFieldAttributes* attribute), 28
 group_by() (*HybridQuery* method), 67
 group_by() (*MultiVectorQuery* method), 84

H

HFCrossEncoderReranker (*class in redisvl.utils.rerank.hf_cross_encoder*), 106
 HFTextVectorizer (*class in redisvl.utils.vectorize.text.huggingface*), 92
 HNSWVectorField (*class in redisvl.schema.fields*), 23
 HNSWVectorFieldAttributes (*class in redisvl.schema.fields*), 24
 hybrid_policy (*VectorQuery* property), 58
 hybrid_policy (*VectorRangeQuery* property), 65
 HybridQuery (*class in redisvl.query*), 65

I

in_order() (*CountQuery* method), 79
 in_order() (*FilterQuery* method), 75
 in_order() (*TextQuery* method), 70
 in_order() (*VectorQuery* method), 54
 in_order() (*VectorRangeQuery* method), 60
 index (*IndexSchema* attribute), 15
 index (*SemanticCache* property), 117
 index_empty (*TagFieldAttributes* attribute), 19
 index_empty (*TextFieldAttributes* attribute), 17
 index_missing (*BaseVectorFieldAttributes* attribute), 22
 IndexSchema (*class in redisvl.schema*), 11
 info() (*AsyncSearchIndex* method), 49
 info() (*SearchIndex* method), 42
 initial_cap (*BaseVectorFieldAttributes* attribute), 22

K

key() (*AsyncSearchIndex* method), 49
 key() (*SearchIndex* method), 42
 key_separator (*AsyncSearchIndex* property), 52
 key_separator (*SearchIndex* property), 44

L

language() (*CountQuery* method), 79
 language() (*FilterQuery* method), 75
 language() (*TextQuery* method), 71
 language() (*VectorQuery* method), 54
 language() (*VectorRangeQuery* method), 60
 limit() (*HybridQuery* method), 67
 limit() (*MultiVectorQuery* method), 85
 limit_fields() (*CountQuery* method), 80
 limit_fields() (*FilterQuery* method), 76

limit_fields() (*TextQuery* method), 71
 limit_fields() (*VectorQuery* method), 55
 limit_fields() (*VectorRangeQuery* method), 60
 limit_ids() (*CountQuery* method), 80
 limit_ids() (*FilterQuery* method), 76
 limit_ids() (*TextQuery* method), 71
 limit_ids() (*VectorQuery* method), 55
 limit_ids() (*VectorRangeQuery* method), 61
 listall() (*AsyncSearchIndex* method), 49
 listall() (*SearchIndex* method), 42
 load() (*AsyncSearchIndex* method), 50
 load() (*HybridQuery* method), 67
 load() (*MultiVectorQuery* method), 85
 load() (*SearchIndex* method), 42

M

m (*HNSWVectorFieldAttributes* attribute), 25
 max_k (*RoutingConfig* attribute), 139
 mdrop() (*EmbeddingsCache* method), 125
 mdrop_by_keys() (*EmbeddingsCache* method), 125
 MessageHistory (class in *redisvl.extensions.message_history*), 132
 messages (*MessageHistory* property), 133
 messages (*SemanticMessageHistory* property), 131
 metadata (*Route* attribute), 139
 mexists() (*EmbeddingsCache* method), 126
 mexists_by_keys() (*EmbeddingsCache* method), 126
 mget() (*EmbeddingsCache* method), 126
 mget_by_keys() (*EmbeddingsCache* method), 127
 min (*DistanceAggregationMethod* attribute), 140
 model_config (*AzureOpenAITextVectorizer* attribute), 97
 model_config (*BaseVectorFieldAttributes* attribute), 22
 model_config (*BedrockTextVectorizer* attribute), 101
 model_config (*CohereReranker* attribute), 106
 model_config (*CohereTextVectorizer* attribute), 100
 model_config (*CustomTextVectorizer* attribute), 103
 model_config (*FlatVectorField* attribute), 30
 model_config (*FlatVectorFieldAttributes* attribute), 31
 model_config (*GeoField* attribute), 21
 model_config (*GeoFieldAttributes* attribute), 21
 model_config (*HFCrossEncoderReranker* attribute), 108
 model_config (*HFTTextVectorizer* attribute), 94
 model_config (*HNSWVectorField* attribute), 24
 model_config (*HNSWVectorFieldAttributes* attribute), 25
 model_config (*IndexSchema* attribute), 15
 model_config (*NumericField* attribute), 19
 model_config (*NumericFieldAttributes* attribute), 20
 model_config (*OpenAITextVectorizer* attribute), 95
 model_config (*Route* attribute), 139
 model_config (*RouteMatch* attribute), 140

model_config (*RoutingConfig* attribute), 139
 model_config (*SemanticRouter* attribute), 138
 model_config (*SVSConfig* attribute), 35
 model_config (*SVSVectorField* attribute), 27
 model_config (*SVSVectorFieldAttributes* attribute), 28
 model_config (*TagField* attribute), 18
 model_config (*TagFieldAttributes* attribute), 19
 model_config (*TextField* attribute), 17
 model_config (*TextFieldAttributes* attribute), 17
 model_config (*Vector* attribute), 53
 model_config (*VertexAITextVectorizer* attribute), 98
 model_config (*VoyageAIReranker* attribute), 109
 model_config (*VoyageAITextVectorizer* attribute), 104
 model_post_init() (*CohereReranker* method), 106
 model_post_init() (*HFCrossEncoderReranker* method), 107
 model_post_init() (*HFTTextVectorizer* method), 93
 model_post_init() (*SemanticRouter* method), 136
 model_post_init() (*VoyageAIReranker* method), 109
 mset() (*EmbeddingsCache* method), 127
 MultiVectorQuery (class in *redisvl.query*), 83

N

name (*AsyncSearchIndex* property), 52
 name (*Route* attribute), 139
 name (*RouteMatch* attribute), 140
 name (*SearchIndex* property), 44
 name (*SemanticRouter* attribute), 138
 no_content() (*CountQuery* method), 80
 no_content() (*FilterQuery* method), 76
 no_content() (*TextQuery* method), 71
 no_content() (*VectorQuery* method), 55
 no_content() (*VectorRangeQuery* method), 61
 no_stem (*TextFieldAttributes* attribute), 17
 no_stopwords() (*CountQuery* method), 80
 no_stopwords() (*FilterQuery* method), 76
 no_stopwords() (*TextQuery* method), 71
 no_stopwords() (*VectorQuery* method), 55
 no_stopwords() (*VectorRangeQuery* method), 61
 Num (class in *redisvl.query.filter*), 89
 NumericField (class in *redisvl.schema.fields*), 19
 NumericFieldAttributes (class in *redisvl.schema.fields*), 20

O

OpenAITextVectorizer (class in *redisvl.utils.vectorize.text.openai*), 94

P

paginate() (*AsyncSearchIndex* method), 50
 paginate() (*SearchIndex* method), 43
 paging() (*CountQuery* method), 80
 paging() (*FilterQuery* method), 76

paging() (*TextQuery* method), 71
 paging() (*VectorQuery* method), 55
 paging() (*VectorRangeQuery* method), 61
 params (*CountQuery* property), 82
 params (*FilterQuery* property), 78
 params (*HybridQuery* property), 68
 params (*MultiVectorQuery* property), 86
 params (*TextQuery* property), 74
 params (*VectorQuery* property), 58
 params (*VectorRangeQuery* property), 65
 phonetic_matcher (*TextFieldAttributes* attribute), 17
 prefix (*AsyncSearchIndex* property), 52
 prefix (*SearchIndex* property), 44

Q

query (*CountQuery* property), 83
 query (*FilterQuery* property), 78
 query (*TextQuery* property), 74
 query (*VectorQuery* property), 59
 query (*VectorRangeQuery* property), 65
 query() (*AsyncSearchIndex* method), 51
 query() (*SearchIndex* method), 43
 query_string() (*CountQuery* method), 80
 query_string() (*FilterQuery* method), 76
 query_string() (*TextQuery* method), 71
 query_string() (*VectorQuery* method), 55
 query_string() (*VectorRangeQuery* method), 61

R

rank() (*CohereReranker* method), 106
 rank() (*HFCrossEncoderReranker* method), 107
 rank() (*VoyageAIReranker* method), 109
 recommend() (*CompressionAdvisor* static method), 33
 reduce (*SVSConfig* attribute), 34
 reduce (*SVSVectorFieldAttributes* attribute), 28
 references (*Route* attribute), 140
 remove_field() (*IndexSchema* method), 14
 remove_route() (*SemanticRouter* method), 137
 return_fields() (*CountQuery* method), 80
 return_fields() (*FilterQuery* method), 76
 return_fields() (*TextQuery* method), 72
 return_fields() (*VectorQuery* method), 55
 return_fields() (*VectorRangeQuery* method), 61
 Route (*class in redisvl.extensions.router*), 139
 route_many() (*SemanticRouter* method), 137
 route_names (*SemanticRouter* property), 138
 route_thresholds (*SemanticRouter* property), 138
 RouteMatch (*class in redisvl.extensions.router.schema*), 140
 routes (*SemanticRouter* attribute), 138
 routing_config (*SemanticRouter* attribute), 138
 RoutingConfig (*class in redisvl.extensions.router*), 139

S

scorer() (*CountQuery* method), 81
 scorer() (*FilterQuery* method), 77
 scorer() (*HybridQuery* method), 68
 scorer() (*MultiVectorQuery* method), 86
 scorer() (*TextQuery* method), 72
 scorer() (*VectorQuery* method), 56
 scorer() (*VectorRangeQuery* method), 62
 search() (*AsyncSearchIndex* method), 51
 search() (*SearchIndex* method), 44
 search_window_size (*SVSConfig* attribute), 35
 search_window_size (*SVSVectorFieldAttributes* attribute), 28
 SearchIndex (*class in redisvl.index*), 37
 SemanticCache (*class in redisvl.extensions.cache.llm*), 110
 SemanticMessageHistory (*class in redisvl.extensions.message_history.semantic_history*), 129
 SemanticRouter (*class in redisvl.extensions.router*), 134
 separator (*TagFieldAttributes* attribute), 19
 set() (*EmbeddingsCache* method), 128
 set_batch_size() (*VectorQuery* method), 56
 set_batch_size() (*VectorRangeQuery* method), 62
 set_client() (*AsyncSearchIndex* method), 52
 set_client() (*SearchIndex* method), 44
 set_distance_threshold() (*VectorRangeQuery* method), 62
 set_ef_runtime() (*VectorQuery* method), 56
 set_epsilon() (*VectorRangeQuery* method), 62
 set_field_weights() (*TextQuery* method), 72
 set_filter() (*CountQuery* method), 81
 set_filter() (*FilterQuery* method), 77
 set_filter() (*TextQuery* method), 72
 set_filter() (*VectorQuery* method), 56
 set_filter() (*VectorRangeQuery* method), 62
 set_hybrid_policy() (*VectorQuery* method), 56
 set_hybrid_policy() (*VectorRangeQuery* method), 63
 set_text_weights() (*HybridQuery* method), 68
 set_text_weights() (*TextQuery* method), 72
 set_threshold() (*SemanticCache* method), 115
 set_ttl() (*EmbeddingsCache* method), 128
 set_ttl() (*SemanticCache* method), 115
 slop() (*CountQuery* method), 81
 slop() (*FilterQuery* method), 77
 slop() (*TextQuery* method), 72
 slop() (*VectorQuery* method), 57
 slop() (*VectorRangeQuery* method), 63
 sort_by() (*CountQuery* method), 81
 sort_by() (*FilterQuery* method), 77
 sort_by() (*HybridQuery* method), 68
 sort_by() (*MultiVectorQuery* method), 86

sort_by() (*TextQuery* method), 73
 sort_by() (*VectorQuery* method), 57
 sort_by() (*VectorRangeQuery* method), 63
 stopwords (*HybridQuery* property), 69
 storage_type (*AsyncSearchIndex* property), 52
 storage_type (*SearchIndex* property), 44
 store() (*MessageHistory* method), 133
 store() (*SemanticCache* method), 115
 store() (*SemanticMessageHistory* method), 131
 sum (*DistanceAggregationMethod* attribute), 140
 SVSConfig (class in *redisvl.utils.compression*), 34
 SVSVectorField (class in *redisvl.schema.fields*), 26
 SVSVectorFieldAttributes (class in *redisvl.schema.fields*), 27

T

Tag (class in *redisvl.query.filter*), 88
 TagField (class in *redisvl.schema.fields*), 18
 TagFieldAttributes (class in *redisvl.schema.fields*), 18
 Text (class in *redisvl.query.filter*), 88
 text_field_name (*TextQuery* property), 74
 text_weights (*HybridQuery* property), 69
 text_weights (*TextQuery* property), 74
 TextField (class in *redisvl.schema.fields*), 16
 TextFieldAttributes (class in *redisvl.schema.fields*), 17
 TextQuery (class in *redisvl.query*), 69
 timeout() (*CountQuery* method), 82
 timeout() (*FilterQuery* method), 78
 timeout() (*TextQuery* method), 73
 timeout() (*VectorQuery* method), 57
 timeout() (*VectorRangeQuery* method), 64
 to_dict() (*IndexSchema* method), 15
 to_dict() (*SemanticRouter* method), 137
 to_yaml() (*IndexSchema* method), 15
 to_yaml() (*SemanticRouter* method), 137
 training_threshold (*SVSVectorFieldAttributes* attribute), 28
 ttl (*EmbeddingsCache* property), 128
 ttl (*SemanticCache* property), 117
 type (*AzureOpenAITextVectorizer* property), 97
 type (*BedrockTextVectorizer* property), 101
 type (*CohereTextVectorizer* property), 100
 type (*CustomTextVectorizer* property), 103
 type (*FlatVectorField* attribute), 31
 type (*GeoField* attribute), 21
 type (*HFTTextVectorizer* property), 94
 type (*HNSWVectorField* attribute), 24
 type (*NumericField* attribute), 20
 type (*OpenAITextVectorizer* property), 95
 type (*SVSVectorField* attribute), 27
 type (*TagField* attribute), 18
 type (*TextField* attribute), 17

type (*VertexAITextVectorizer* property), 98
 type (*VoyageAITextVectorizer* property), 104

U

unf (*NumericFieldAttributes* attribute), 20
 unf (*TextFieldAttributes* attribute), 17
 update() (*SemanticCache* method), 116
 update_route_thresholds() (*SemanticRouter* method), 138
 update_routing_config() (*SemanticRouter* method), 138
 uppercase_strings() (*BaseVectorFieldAttributes* class method), 22

V

validate_svs_params() (*SVSVectorFieldAttributes* method), 27
 validate_vector() (*Vector* method), 52
 Vector (class in *redisvl.query*), 52
 vectorizer (*SemanticRouter* attribute), 138
 VectorQuery (class in *redisvl.query*), 53
 VectorRangeQuery (class in *redisvl.query*), 59
 verbatim() (*CountQuery* method), 82
 verbatim() (*FilterQuery* method), 78
 verbatim() (*TextQuery* method), 73
 verbatim() (*VectorQuery* method), 58
 verbatim() (*VectorRangeQuery* method), 64
 version (*IndexSchema* attribute), 15
 VertexAITextVectorizer (class in *redisvl.utils.vectorize.text.vertexai*), 97
 VoyageAIReranker (class in *redisvl.utils.rerank.voyageai*), 108
 VoyageAITextVectorizer (class in *redisvl.utils.vectorize.text.voyageai*), 103

W

weight (*TextFieldAttributes* attribute), 17
 with_payloads() (*CountQuery* method), 82
 with_payloads() (*FilterQuery* method), 78
 with_payloads() (*TextQuery* method), 74
 with_payloads() (*VectorQuery* method), 58
 with_payloads() (*VectorRangeQuery* method), 64
 with_schema() (*HybridQuery* method), 68
 with_schema() (*MultiVectorQuery* method), 86
 with_scores() (*CountQuery* method), 82
 with_scores() (*FilterQuery* method), 78
 with_scores() (*TextQuery* method), 74
 with_scores() (*VectorQuery* method), 58
 with_scores() (*VectorRangeQuery* method), 64
 withsuffixtrie (*TagFieldAttributes* attribute), 19
 withsuffixtrie (*TextFieldAttributes* attribute), 18