pymilvus

Release 1.1.0

Milvus

Jun 09, 2021
# TABLE OF CONTENTS

1 Overview

1.1 Installation ................................................................. 1
  1.1.1 Installing via pip .................................................. 1
  1.1.2 Installing in a virtual environment .............................. 1
  1.1.3 Installing a specific PyMilvus version ....................... 2
  1.1.4 Installing from source ......................................... 2
  1.1.5 Verifying installation ......................................... 2

1.2 Tutorial ................................................................. 2
  1.2.1 Prerequisites ...................................................... 3
  1.2.2 Connect to Milvus ................................................ 3
  1.2.3 Collection ......................................................... 3
  1.2.4 Create Collection ................................................ 3
  1.2.5 Create Partition ................................................ 4
  1.2.6 Entities ............................................................. 5
  1.2.7 Insert Entities ...................................................... 5
  1.2.8 Flush ................................................................. 5
  1.2.9 Get Detailed information ..................................... 5
  1.2.10 Count Entities ..................................................... 6
  1.2.11 Get ................................................................. 6
    1.2.11.1 Get Entities by ID ......................................... 6
  1.2.12 Search ............................................................. 6
    1.2.12.1 Search Entities by Vector Similarity ................... 6
  1.2.13 Deletion ........................................................... 7
    1.2.13.1 Delete Entities by id .................................... 7
    1.2.13.2 Drop a Partition ........................................... 7
    1.2.13.3 Drop a Collection ........................................ 8

1.3 API reference .......................................................... 8
  1.3.1 Client .............................................................. 8
    1.3.1.1 Constructor .................................................. 8
    1.3.1.2 Methods ....................................................... 8
    1.3.1.3 APIs .......................................................... 8
  1.3.2 Index Type .......................................................... 15
  1.3.3 Metric Type ........................................................ 15

1.4 Index ................................................................. 16
  1.4.1 Vector Index ........................................................ 16
    1.4.1.1 FLAT ............................................................ 16
    1.4.1.2 IVF_FLAT ...................................................... 17
    1.4.1.3 IVF_PQ .......................................................... 17
    1.4.1.4 IVF_SQ8 ........................................................ 18
    1.4.1.5 IVF_SQ8_H ..................................................... 19
PyMilvus is a python SDK for Milvus and is a recommended way to work with Milvus. This documentation covers every thing you need to know about PyMilvus.

**Installation**  Instructions on how to install PyMilvus.

**Tutorial**  A quick start to use PyMilvus.

**API reference**  The complete API documentation.

**Index**  Index and relevant parameters.

**Search results**  How to deal with search results.

**Changelog**  Changes in the latest PyMilvus.

**Contributing**  Method of contribution, bug shooting and contribution guide.

**FAQ**  Some questions that come up often.

**About this documentation**  How this documentation is generated.

### 1.1 Installation

#### 1.1.1 Installing via pip

PyMilvus is in the [Python Package Index](https://pypi.org).

PyMilvus only support python3(>= 3.6), usually, it’s ok to install PyMilvus like below.

```bash
$ python3 -m pip install pymilvus
```

#### 1.1.2 Installing in a virtual environment

It’s recommended to use PyMilvus in a virtual environment, using virtual environment allows you to avoid installing Python packages globally which could break system tools or other projects. We use [virtualenv](https://virtualenv.pypa.io) as an example to demonstrate how to install and using PyMilvus in a virtual environment. See [virtualenv](https://virtualenv.pypa.io) for more information about why and how.

```bash
$ python3 -m pip install virtualenv
$ virtualenv venv
$ source venv/bin/activate
(venv) $ pip install pymilvus
```
If you want to exit the virtual environment, you can use `deactivate`.

```bash
(venv) $ deactivate
```

### 1.1.3 Installing a specific PyMilvus version

Here we assume you are already in a virtual environment. Suitable PyMilvus version depends on Milvus version you are using. See `install pymilvus` for recommended pymilvus version.

If you want to install a specific version of PyMilvus:

```bash
(venv) $ pip install pymilvus==1.1.0
```

If you want to upgrade PyMilvus into the latest version published:

```bash
(venv) $ pip install --upgrade pymilvus
```

### 1.1.4 Installing from source

This will install the latest PyMilvus into your virtual environment.

```bash
(venv) $ pip install git+https://github.com/milvus-io/pymilvus.git
```

### 1.1.5 Verifying installation

Your installation is correct if the following command in the Python shell doesn’t raise an exception.

```bash
(venv) $ python -c "from milvus import Milvus, DataType"
```

*Section author: Yangxuan@milvus*

### 1.2 Tutorial

This is a basic introduction to Milvus by PyMilvus.

For a runnable python script, checkout example.py on PyMilvus Github, or hello milvus on Milvus official website. It’s a good recommended start to get started with Milvus and PyMilvus as well.

*Note: Here we use float vectors as example vector field data, if you want to learn example about binary vectors, see binary vector example.*
1.2.1 Prerequisites

Before we start, there are some prerequisites.

Make sure that:

- You have a running Milvus instance.
- PyMilvus is correctly installed.

1.2.2 Connect to Milvus

First of all, we need to import PyMilvus.

```python
>>> from milvus import Milvus, DataType, MetricType
```

Then, we can make connection with Milvus server. By default Milvus runs on localhost in port 19530, so you can use default value to connect to Milvus.

```python
>>> host = '127.0.0.1'
>>> port = '19530'
>>> client = Milvus(host, port)
```

After connecting, we can communicate with Milvus in the following ways. If you are confused about the terminology, see Milvus Terminology for explanations.

1.2.3 Collection

Now let’s create a new collection. Before we start, we can list all the collections already exist. For a brand new Milvus running instance, the result should be empty.

```python
>>> client.list_collections()
(Status(code=0, message='Show collections successfully!'), [])
```

1.2.4 Create Collection

To create collection, we need to provide collection parameters. `collection_param` consists of 4 components, they are `collection_name`, `dimension`, `index_file_size` and `metric_type`.

- **collection_name**: The name of collection should be a unique string to collections already exist.
- **dimension**: For a float vector, dimension should be equal to the length of a vector; for a binary vector, dimension should be equal to bit size of a vector.
- **index_file_size**: Milvus controls the size of data segment according to the `index_file_size`, you can refer to Storage Concepts for more information about segments and `index_file_size`.
- **metric_type**: Milvus compute distance between two vectors, you can refer to Distance Metrics for more information.

Now we can create a collection:

```python
>>> collection_name = 'demo_film_tutorial'
>>> collection_param = {
...     "collection_name": collection_name,
...     "dimension": 8,
...     "index_file_size": 2048,
...}
```
... "metric_type": MetricType.L2
... }
>>> client.create_collection(collection_param)
Status(code=0, message='Create collection successfully!')

Then you can list collections and ‘demo_film_tutorial’ will be in the result.

>>> client.list_collections()
(Status(code=0, message='Show collections successfully!'), ['demo_film_tutorial'])

You can also get info of the collection.

>>> status, info = client.get_collection_info(collection_name)
>>> info
CollectionSchema(collection_name='demo_film_tutorial', dimension=8, index_file_size=2048, metric_type=<MetricType: L2>)

The attributes of collection can be extracted from info.

>>> info.collection_name
'demo_film_tutorial'

>>> info.dimension
8

>>> info.index_file_size
2048

>>> info.metric_type
<MetricType: L2>

This tutorial is a basic intro tutorial, building index won’t be covered by this tutorial. If you want to go further into Milvus with indexes, it’s recommended to check our index examples.

If you’re already known about indexes from index examples, and you want a full lists of params supported by PyMilvus, you check out Index chapter of the PyMilvus documentation.

Further more, if you want to get a thorough view of indexes, check our official website for Vector Index.

1.2.5 Create Partition

If you don’t create a partition, there will be a default one called “_default”, all the entities will be inserted into the “_default” partition. You can check it by list_partitions()

>>> client.list_partitions(collection_name)
(Status(code=0, message='Success'), [{collection_name='demo_film_tutorial', tag='_default'}])

You can provide a partition tag to create a new partition.

>>> client.create_partition(collection_name, "films")
Status(code=0, message='OK')

>>> client.list_partitions(collection_name)
(Status(code=0, message='Success'), [{collection_name='demo_film_tutorial', tag='_default'}, {collection_name='demo_film_tutorial', tag='films'}])
1.2.6 Entities

An entity is a group of fields that corresponds to real world objects. In current version, Milvus only contains a vector field. Here is an example of 3 entities structured in list of list.

```python
>>> import random
>>> entities = [[random.random() for _ in range(8)] for _ in range(3)]
```

1.2.7 Insert Entities

```python
>>> status, ids = client.insert(collection_name, entities)
```

If the entities inserted successfully, `ids` we provided will be returned.

```python
>>> ids
[1615279498011637000, 1615279498011637001, 1615279498011637002]
```

Or you can also provide entity ids

```python
>>> entity_ids = [0, 1, 2]
>>> status, ids = client.insert(collection_name, entities, entity_ids)
```

**Warning:** If the first time when `insert()` is invoked `ids` is not passed into this method, each of the rest time when `insert()` is invoked `ids` is not permitted to pass, otherwise server will return an error and the insertion process will fail. And vice versa.

1.2.8 Flush

After successfully inserting 3 entities into Milvus, we can Flush data from memory to disk so that we can retrieve them. Milvus also performs an automatic flush with a fixed interval(configurable, default 1 second), see Data Flushing.

You can flush multiple collections at one time, so be aware the parameter is a list.

```python
>>> client.flush([collection_name])
Status(code=0, message='OK')
```

1.2.9 Get Detailed information

After insertion, we can get the detail of collection statistics information by `get_collection_stats()`

**Note:** For a better output format, we are using `pprint` to provide a better format.

```python
>>> from pprint import pprint
>>> status, stats = client.get_collection_stats(collection_name)
>>> pprint(stats)
{'partitions': [{'row_count': 3, 'segments': [{'data_size': 120, 'index_name': 'IDMAP', 'name': '1615279498038473000'}, ...]
```
1.2.10 Count Entities

We can also count how many entities are there in the collection.

```python
>>> client.count_entities(collection_name)
(Status(code=0, message='Success!'), 3)
```

1.2.11 Get

1.2.11.1 Get Entities by ID

You can get entities by their ids.

```python
>>> status, films = client.get_entity_by_id(collection_name, [0, 1615279498011637001])
>>> films
[[], [0.8309633731842041, 0.7896093726158142, 0.09463301301002502, 0.7827594876289368, ...
```

If id exists, an entity will be returned. If id doesn’t exist, [] will be return. For the example above, the result films will only have one entity, the other is []. Because vector id are generated by server, so the value of id may differ.

1.2.12 Search

1.2.12.1 Search Entities by Vector Similarity

You can get entities by vector similarity. Assuming we have a film_A like below, and we want to get top 2 films that are most similar with it.

```python
>>> film_A = [random.random() for _ in range(8)]
>>> status, results = client.search(collection_name, 2, [film_A])
```

Note: If the collection is index-built, user need to specify search param, and pass parameter params like: `client.search(...., params={...})`. You can refer to Index params for more details.

Note: If parameter partition_tags is specified, milvus executes search request on these partition instead of whole collection.

The returned results is a 2-D like structure, 1 for 1 entity querying, 2 for top 2. For more clarity, we obtain the film as below. If you want to know how to deal with search result in a better way, you can refer to search result in PyMilvus doc.
>>> result = results[0]
>>> film_1 = result[0]
>>> film_2 = result[1]

Then how do we get ids, distances and fields? It’s as below.

Note: Because vectors are randomly generated, so the retrieved vector id and distance may differ.

>>> film_1.id  # id
1615279498011637002

>>> film_1.distance  # distance
1.0709768533706665

1.2.13 Deletion

Finally, let’s move on to deletion in Milvus. We can delete entities by ids, drop a whole partition, or drop the entire collection.

1.2.13.1 Delete Entities by id

You can delete entities by their ids.

>>> client.delete_entity_by_id(collection_name, [0, 1615279498011637002])
Status(code=0, message='OK')

Note: If one entity corresponding to a specified id doesn’t exist, milvus ignore it and execute next deletion. In this case, client always return ok status except any exception occurs.

>>> client.count_entities(collection_name)
(Status(code=0, message='Success!'), 2)

1.2.13.2 Drop a Partition

You can also drop a partition.

Danger: Once you drop a partition, all the data in this partition will be deleted too.

>>> client.drop_partition(collection_name, "films")
Status(code=0, message='OK')
1.2.13.3 Drop a Collection

Finally, you can drop an entire collection.

**Danger:** Once you drop a collection, all the data in this collection will be deleted too.

```python
>>> client.drop_collection(collection_name)
Status(code=0, message='OK')
```

Section author: Yangxuan@milvus

1.3 API reference

1.3.1 Client

1.3.1.1 Constructor

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Milvus()</td>
<td>Milvus client</td>
</tr>
</tbody>
</table>

1.3.1.2 Methods

1.3.1.3 APIs

```python
class milvus.Milvus (host=None, port=None, handler='GRPC', pool='SingletonThread', **kwargs)
```

```python
create_collection (param, timeout=30)
```

Creates a collection.

**Parameters**

- **param** (dict) – Information needed to create a collection. It contains items:
  - `collection_name` (str) – Collection name.
  - `dimension` (int) – Dimension of embeddings stored in collection.
  - `index_file_size` (int) – Segment size. See Storage Concepts.
  - `metric_type` (MetricType) – Distance Metrics type. Valued form MetricType. See Distance Metrics.

A demo is as follow:

```python
param={'collection_name': 'name',
       'dimension': 16,
       'index_file_size': 1024 # Optional, default 1024
       'metric_type': MetricType.L2 # Optional, default MetricType.L2
}
```

- **timeout** (float) – An optional duration of time in seconds to allow for the RPC. When timeout is set to None, client waits until server responses or error occurs.
has_collection(collection_name, timeout=30)
Checks whether a collection exists.

Parameters

• **collection_name** *(str)* – Name of the collection to check.

• **timeout** *(float)* – An optional duration of time in seconds to allow for the RPC. When timeout is set to None, client waits until server responses or error occurs.

Returns The operation status and the flag indicating if collection exists. Succeed if Status.OK() is True. If status is not OK, the flag is always False.

Return type Status, bool

get_collection_info(collection_name, timeout=30)
Returns information of a collection.

Parameters

• **collection_name** *(str)* – Name of the collection to describe.

• **timeout** *(float)* – An optional duration of time in seconds to allow for the RPC. When timeout is set to None, client waits until server responses or error occurs.

Returns The operation status and collection information. Succeed if Status.OK() is True. If status is not OK, the returned information is always None.

Return type Status, CollectionSchema

count_entities(collection_name, timeout=30)
Returns the number of vectors in a collection.

Parameters

• **collection_name** *(str)* – target table name.

• **timeout** *(float)* – An optional duration of time in seconds to allow for the RPC. When timeout is set to None, client waits until server responses or error occurs.

Returns The operation status and row count. Succeed if Status.OK() is True. If status is not OK, the returned value of is always None.

Return type Status, int

list_collections(timeout=30)
Returns collection list.

Parameters **timeout** *(float)* – An optional duration of time in seconds to allow for the RPC. When timeout is set to None, client waits until server responses or error occurs.

Returns The operation status and collection name list. Succeed if Status.OK() is True. If status is not OK, the returned name list is always [].

Return type Status, list[str]

get_collection_stats(collection_name, timeout=30)
Returns collection statistics information.

Parameters

• **collection_name** *(str)* – target table name.
timeout (float) – An optional duration of time in seconds to allow for the RPC. When timeout is set to None, client waits until server responses or error occurs.

Returns The operation status and collection statistics information. Succeed if Status.OK() is True. If status is not OK, the returned information is always [].

Return type Status, dict

load_collection (collection_name, partition_tags=None, timeout=None)
Loads a collection for caching.

Parameters
- collection_name (str) – collection to load
- partition_tags – partition tag list. None indicates to load whole collection, otherwise to load specified partitions.
- timeout (float) – An optional duration of time in seconds to allow for the RPC. When timeout is set to None, client waits until server responses or error occurs.

Returns The operation status. Succeed if Status.OK() is True.

Return type Status

release_collection (collection_name, partition_tags=None, timeout=None)
Release a collection from memory and cache.

Parameters
- collection_name (str) – collection to release
- partition_tags – partition tag list. None indicates to release whole collection, otherwise to release specified partitions.
- timeout (float) – An optional duration of time in seconds to allow for the RPC. When timeout is set to None, client waits until server responses or error occurs.

Returns The operation status. Succeed if Status.OK() is True.

Return type Status

drop_collection (collection_name, timeout=30)
Deletes a collection by name.

Parameters
- collection_name (str) – Name of the collection being deleted
- timeout (float) – An optional duration of time in seconds to allow for the RPC. When timeout is set to None, client waits until server responses or error occurs.

Returns The operation status. Succeed if Status.OK() is True.

Return type Status

insert (collection_name, records, ids=None, partition_tag=None, params=None, timeout=None, **kwargs)
Insert vectors to a collection.

Parameters
- collection_name (str) – Name of the collection to insert vectors to.
- ids (list [int]) – ID list. None indicates ID is generated by server system. Note that if the first time when insert() is invoked ids is not passed into this method, each of the rest
time when `inset()` is invoked, ids is not permitted to pass, otherwise server will return an error and the insertion process will fail. And vice versa.

- **records** (`list[list[float]]`) – List of vectors to insert.
- **partition_tag** (`str or None`, If partition_tag is None, vectors will be inserted to the default partition `_default`) – Tag of a partition.
- **params** (`dict`) – Insert param. Reserved.
- **timeout** (`float`) – An optional duration of time in seconds to allow for the RPC. When timeout is set to None, client waits until server responses or error occurs.
- **kwargs** –
  - `_async` (`bool`) – Indicate if invoke asynchronously. When value is true, method returns a InsertFuture object; otherwise, method returns results from server.
  - `_callback` (`function`) – The callback function which is invoked after server response successfully. It only takes effect when `_async` is set to True.

**Returns** The operation status and IDs of inserted entities. Succeed if `Status.OK()` is True. If status is not OK, the returned IDs is always `[]`.

**Return type** `Status, list[int]`

**get_entity_by_id** (`collection_name`, `ids`, `timeout=None, partition_tag=None`)
Returns raw vectors according to ids.

**Parameters**
- **collection_name** (`str`) – Name of the collection
- **ids** (`list`) – list of vector id
- **timeout** (`float`) – An optional duration of time in seconds to allow for the RPC. When timeout is set to None, client waits until server responses or error occurs.
- **partition_tag** (`str`) – The partition tag of entity

**Returns** The operation status and entities. Succeed if `Status.OK()` is True. If status is not OK, the returned entities is always `[]`.

**Return type** `Status, list[list[float]]`

**list_id_in_segment** (`collection_name`, `segment_name`, `timeout=None`)
Get IDs of entity stored in the specified segment.

**Parameters**
- **collection_name** (`str`) – Collection the segment belongs to.
- **segment_name** (`str`) – Segment name.
- **timeout** (`float`) – An optional duration of time in seconds to allow for the RPC. When timeout is set to None, client waits until server responses or error occurs.

**Returns** The operation status and entity IDs. Succeed if `Status.OK()` is True. If status is not OK, the returned IDs is always `[]`.

**Return type** `Status, list[int]`

**create_index** (`collection_name`, `index_type=None`, `params=None`, `timeout=None`, **kwargs)
Creates index for a collection.

**Parameters**
- `collection_name (str)` – Collection used to create index.
- `index_type (IndexType)` – Index params. See index params for supported indexes.
- `params (dict)` – Index param. See index params for detailed index param of supported indexes.
- `timeout (float)` – An optional duration of time in seconds to allow for the RPC. When timeout is set to None, client waits until server responses or error occurs.
- `kwargs` –
  - `_async (bool)` – Indicate if invoke asynchronously. When value is true, method returns a IndexFuture object; otherwise, method returns results from server.
  - `_callback (function)` – The callback function which is invoked after server response successfully. It only takes effect when _async is set to True.

  **Returns** The operation status. Succeed if `Status.OK()` is True.

  **Return type** Status

  `get_index_info (collection_name, timeout=30)`

  Show index information of a collection.

  **Parameters**
  - `collection_name (str)` – table name been queried
  - `timeout (float)` – An optional duration of time in seconds to allow for the RPC. When timeout is set to None, client waits until server responses or error occurs.

  **Returns** The operation status and index info. Succeed if `Status.OK()` is True. If status is not OK, the returned index info is always None.

  **Return type** Status, IndexParam

  `drop_index (collection_name, timeout=30)`

  Removes an index.

  **Parameters**
  - `collection_name (str)` – target collection name.
  - `timeout (float)` – An optional duration of time in seconds to allow for the RPC. When timeout is set to None, client waits until server responses or error occurs.

  **Returns** The operation status. Succeed if `Status.OK()` is True.

  **Return type** Status

  `create_partition (collection_name, partition_tag, timeout=30)`

  create a partition for a collection.

  **Parameters**
  - `collection_name (str)` – Name of the collection.
  - `partition_tag (str)` – Name of the partition tag.
  - `timeout (float)` – An optional duration of time in seconds to allow for the RPC. When timeout is set to None, client waits until server responses or error occurs.

  **Returns** The operation status. Succeed if `Status.OK()` is True.

  **Return type** Status
has_partition (collection_name, partition_tag, timeout=30)

Check if specified partition exists.

Parameters

- collection_name (str) – target table name.
- partition_tag (str) – partition tag.
- timeout (float) – An optional duration of time in seconds to allow for the RPC. When timeout is set to None, client waits until server responses or error occurs.

Returns The operation status and a flag indicating if partition exists. Succeed if Status.OK() is True. If status is not ok, the flag is always False.

Return type Status, bool

list_partitions (collection_name, timeout=30)

Show all partitions in a collection.

Parameters

- collection_name (str) – target table name.
- timeout (float) – An optional duration of time in seconds to allow for the RPC. When timeout is set to None, client waits until server responses or error occurs.

Returns The operation status and partition list. Succeed if Status.OK() is True. If status is not OK, returned partition list is [].

Return type Status, list[PartitionParam]

drop_partition (collection_name, partition_tag, timeout=30)

Deletes a partition in a collection.

Parameters

- collection_name (str) – Collection name.
- partition_tag (str) – Partition name.
- timeout (float) – An optional duration of time in seconds to allow for the RPC. When timeout is set to None, client waits until server responses or error occurs.

Returns The operation status. Succeed if Status.OK() is True.

Return type Status

search (collection_name, top_k, query_records, partition_tags=None, params=None, timeout=None, **kwargs)

Search vectors in a collection.

Parameters

- collection_name (str) – Name of the collection.
- top_k (int) – number of vectors which is most similar with query vectors
- query_records (list[list[float32]]) – vectors to query
- partition_tags (list) – tags to search. None indicates to search in whole collection.
- params (dict) – Search params. The params is related to index type the collection is built. See index params for more detailed information.
- timeout (float) – An optional duration of time in seconds to allow for the RPC. When timeout is set to None, client waits until server responses or error occurs.
• **kwargs –
  – `_async` (bool) – Indicate if invoke asynchronously. When value is true, method returns a SearchFuture object; otherwise, method returns results from server.
  – `_callback` (function) – The callback function which is invoked after server response successfully. It only takes effect when `_async` is set to True.

Returns The operation status and search result. See <a>here</a> to find how to handle search result. Succeed if `Status.OK()` is True. If status is not OK, results is always None.

Return type Status, TopKQueryResult

**delete_entity_by_id**(collection_name, id_array, timeout=None, partition_tag=None)
Deletes vectors in a collection by vector ID.

Parameters

• **collection_name** (str) – Name of the collection.
• **id_array** (list[int]) – list of vector id
• **timeout** (float) – An optional duration of time in seconds to allow for the RPC. When timeout is set to None, client waits until server responses or error occurs.
• **partition_tag** (str) – The partition tag of entity

Returns The operation status. If the specified ID doesn’t exist, Milvus server skip it and try to delete next entities, which is regard as one successful operation. Succeed if `Status.OK()` is True.

Return type Status

**flush**(collection_name_array=None, timeout=None, **kwargs)
Flushes vector data in one collection or multiple collections to disk.

Parameters

• **collection_name_array** (list) – Name of one or multiple collections to flush.
• **timeout** (float) – An optional duration of time in seconds to allow for the RPC. When timeout is set to None, client waits until server responses or error occurs.
• **kwargs** –
  – `_async` (bool) – Indicate if invoke asynchronously. When value is true, method returns a FlushFuture object; otherwise, method returns results from server.
  – `_callback` (function) – The callback function which is invoked after server response successfully. It only takes effect when `_async` is set to True.

Returns The operation status. Succeed if `Status.OK()` is True.

Return type Status

**compact**(collection_name, timeout=None, **kwargs)
Compacts segments in a collection. This function is recommended after deleting vectors.

Parameters

• **collection_name** (str) – Name of the collections to compact.
• **timeout** (float) – An optional duration of time in seconds to allow for the RPC. When timeout is set to None, client waits until server responses or error occurs.
• **kwargs** –
- \_async (bool) – Indicate if invoke asynchronously. When value is true, method returns a CompactFuture object; otherwise, method returns results from server.

- \_callback (function) – The callback function which is invoked after server response successfully. It only takes effect when \_async is set to True.

**Returns** The operation status. Succeed if Status.OK() is True.

**Return type** Status

### 1.3.2 Index Type

class milvus\_IndexType

Index type enum.

**INVALID** = 0

Invalid index type.

**FLAT** = 1

FLAT index. See FLAT.

**IVF\_FLAT** = 2

IVF(Inverted File) FLAT index. See IVF\_FLAT.

**IVF\_SQ8** = 3

IVF SQ8 index. See IVF\_SQ8.

**RNSG** = 4

RNSG(Refined NSG) index. See RNSG.

**IVF\_SQ8H** = 5

IVF SQ8 Hybrid index. See IVF\_SQ8H.

**IVF\_PQ** = 6

IVF PQ index. See IVF\_PQ.

**HNSW** = 11

HNSW index. See HNSW.

**ANNOY** = 12

ANNOY index. See ANNOY.

**IV\_FLAT** = 2

Alternative name for IVF\_FLAT. Reserved for compatibility.

**IVF\_SQ8\_H** = 5

Alternative name for IVF\_SQ8H. Reserved for compatibility.

### 1.3.3 Metric Type

class milvus\_MetricType

Metric type enum.

**INVALID** = 0

Invalid metric type.

**L2** = 1


**IP** = 2

Inner product. A metric for float vectors. See Inner Product.
HAMMING = 3

JACCARD = 4

TANIMOTO = 5
Tanimoto distance. A metric for binary vectors. See Tanimoto distance.

SUBSTRUCTURE = 6
Superstructure. A metric for binary vectors, only support FLAT index. See Superstructure.

SUPERSTRUCTURE = 7
Substructure. A metric for binary vectors, only support FLAT index. See Substructure.

1.4 Index

Milvus support to create index to accelerate vector approximate search.

To learn how to create an index by python client, see method create_index() and index example.

For more detailed information about indexes, please refer to Milvus documentation index chapter.

To learn how to choose an appropriate index for your application scenarios, please read How to Select an Index in Milvus.

To learn how to choose an appropriate index for a metric, see Distance Metrics.

1.4.1 Vector Index

- FLAT
- IVF_FLAT
- IVF_SQ8
- IVF_SQ8_H
- IVF_PQ
- HNSW
- ANNOY
- RNSG

1.4.1.1 FLAT

If FLAT index is used, the vectors are stored in an array of float/binary data without any compression. during searching vectors, all indexed vectors are decoded sequentially and compared to the query vectors.

FLAT index provides 100% query recall rate. Compared to other indexes, it is the most efficient indexing method when the number of queries is small.

The inserted and index-inbuilt vectors and index-dropped vectors are regard as built with FLAT.

- building parameters: N/A

```python
# FLAT
client.create_index(collection_name, IndexType.FLAT)
```
• search parameters: N/A

```python
# FLAT
client.search(collection_name,
   1,
   query_vectors
)
```

1.4.1.2 IVF_FLAT

IVF (Inverted File) is an index type based on quantization. It divides the points in space into \( nlist \) units by clustering method. During searching vectors, it compares the distances between the target vector and the center of all the units, and then select the \( nprobe \) nearest unit. Then, it compares all the vectors in these selected cells to get the final result.

IVF_FLAT is the most basic IVF index, and the encoded data stored in each unit is consistent with the original data.

• building parameters:
  
  \( nlist \): Number of cluster units.

```python
# IVF_FLAT
client.create_index(collection_name, IndexType.IVF_FLAT, {
   "nlist": 100 # int. 1~65536
})
```

• search parameters:
  
  \( nprobe \): Number of inverted file cell to probe.

```python
# IVF_FLAT
client.search(collection_name,
   1,
   query_vectors,
   params={
   "nprobe": 8 # int. 1\~nlist(cpu), 1\~min[2048, nlist](gpu)
   }
)
```

1.4.1.3 IVF_PQ

PQ (Product Quantization) uniformly decomposes the original high-dimensional vector space into Cartesian products of \( m \) low-dimensional vector spaces, and then quantizes the decomposed low-dimensional vector spaces. In the end, each vector is stored in \( m \times nbits \) bits. Instead of calculating the distances between the target vector and the center of all the units, product quantization enables the calculation of distances between the target vector, and the clustering center of each low-dimensional space and greatly reduces the time complexity and space complexity of the algorithm.

IVF_PQ performs IVF index clustering, and then quantizes the product of vectors. Its index file is even smaller than IVF_SQ8, but it also causes a loss of accuracy during searching.

• building parameters:
  
  \( nlist \): Number of cluster units.

  \( m \): Number of factors of product quantization. **CPU-only Milvus:** \( m \mod \dim \); **GPU-enabled Milvus:** \( m \{1, 2, 3, 4, 8, 12, 16, 20, 24, 28, 32, 40, 48, 56, 64, 96\} \), and \( (\dim / m) \{ 1, 2, 3, 4, 6, 8, 10, 12, 16, 20, 24, 28, 32 \} \). (\( m \times 1024 \) \( \text{MaxSharedMemPerBlock} \) of your graphics card).

  \( nbits \): Number of bits in which each low-dimensional vector is stored.
client.create_index(collection_name, 
    IndexType.IVF_PQ, 
    { 
        "nlist": 100,  # int. 1~65536 
        "m": 8        # int. 1~16. 8 by default 
    })

• search parameters:
  
  nprobe: Number of inverted file cell to probe.

client.search(collection_name, 
1, 
query_vectors, 
params={ 
    "nprobe": 8  # int. 1-nlist(cpu), 1-min[2048, nlist](gpu) 
})

1.4.1.4 IVF_SQ8

IVF_SQ8 does scalar quantization for each vector placed in the unit based on IVF. Scalar quantization converts each dimension of the original vector from a 4-byte floating-point number to a 1-byte unsigned integer, so the IVF_SQ8 index file occupies much less space than the IVF_FLAT index file. However, scalar quantization results in a loss of accuracy during searching vectors.

• building parameters:
  
  nlist: Number of cluster units.

client.create_index(collection_name, 
    IndexType.IVF_SQ8, 
    { 
        "nlist": 100  # int. 1~65536 
    })

• search parameters:
  
  nprobe: Number of inverted file cell to probe.

client.search(collection_name, 
1, 
query_vectors, 
params={ 
    "nprobe": 8  # int. 1-nlist(cpu), 1-min[2048, nlist](gpu) 
})
1.4.1.5 IVF SQ8_H

Optimized version of IVF SQ8 that requires both CPU and GPU to work. Unlike IVF SQ8, IVF SQ8_H uses a GPU-based coarse quantizer, which greatly reduces time to quantize.

IVF SQ8H is an IVF SQ8 index that optimizes query execution.

The query method is as follows:

- If \( nq \geq \text{gpu_search_threshold} \), GPU handles the entire query task.
- If \( nq < \text{gpu_search_threshold} \), GPU handles the task of retrieving the \( nprobe \) nearest unit in the IVF index file, and CPU handles the rest.

- **building parameters:**
  
  - **nlist**: Number of cluster units.

  ```python
  # IVF SQ8_H
  client.create_index(collection_name,
                      IndexType.IVF_SQ8_H,
                      {
                        "nlist": 100  # int. 1~65536
                      })
  ```

- **search parameters:**
  
  - **nprobe**: Number of inverted file cell to probe.

  ```python
  # IVF SQ8_H
  client.search(collection_name,
                1,
                query_vectors,
                params={
                  "nprobe": 8  # int. 1-nlist(cpu), 1-min[2048, nlist](gpu)
                })
  ```

1.4.1.6 ANNOY

ANNOY (Approximate Nearest Neighbors Oh Yeah) is an index that uses a hyperplane to divide a high-dimensional space into multiple subspaces, and then stores them in a tree structure.

When searching for vectors, ANNOY follows the tree structure to find subspaces closer to the target vector, and then compares all the vectors in these subspaces (The number of vectors being compared should not be less than \( \text{search}_k \)) to obtain the final result. Obviously, when the target vector is close to the edge of a certain subspace, sometimes it is necessary to greatly increase the number of searched subspaces to obtain a high recall rate. Therefore, ANNOY uses \( n\_trees \) different methods to divide the whole space, and searches all the dividing methods simultaneously to reduce the probability that the target vector is always at the edge of the subspace.

- **building parameters:**
  
  - **n_trees**: The number of methods of space division.

  ```python
  # ANNOY
  client.create_index(collection_name,
                      IndexType.ANNOY,
                      {
                      })
  ```

(continues on next page)
search parameters:

search_k: The number of nodes to search. -1 means 5% of the whole data.

client.search(collection_name, 1, query_vectors, params={
  "search_k": -1  # int. (-1) U [top_k, n*n_trees], n represents vectors count.
})

1.4.1.7 HNSW

HNSW (Hierarchical Navigable Small World Graph) is a graph-based indexing algorithm. It builds a multi-layer navigation structure for an image according to certain rules. In this structure, the upper layers are more sparse and the distances between nodes are farther; the lower layers are denser and the distances between nodes are closer. The search starts from the uppermost layer, finds the node closest to the target in this layer, and then enters the next layer to begin another search. After multiple iterations, it can quickly approach the target position.

In order to improve performance, HNSW limits the maximum degree of nodes on each layer of the graph to $M$. In addition, you can use efConstruction (when building index) or ef (when searching targets) to specify a search range.

building parameters:

M: Maximum degree of the node.

efConstruction: Take the effect in stage of index construction.

client.create_index(collection_name, IndexType.HNSW, {
  "M": 16,  # int. 4~64
  "efConstruction": 40  # int. 8~512
})

search parameters:

ef: Take the effect in stage of search scope, should be larger than top_k.

client.search(collection_name, 1, query_vectors, params={
  "ef": 64  # int. top_k=32768
})
### 1.4.1.8 RNSG

**RNSG** *(Refined Navigating Spreading-out Graph)* is a graph-based indexing algorithm. It sets the center position of the whole image as a navigation point, and then uses a specific edge selection strategy to control the out-degree of each point (less than or equal to `out_degree`). Therefore, it can reduce memory usage and quickly locate the target position nearby during searching vectors.

The graph construction process of NSG is as follows:

1. Find `knng` nearest neighbors for each point.
2. Iterate at least `search_length` times based on `knng` nearest neighbor nodes to select `candidate_pool_size` possible nearest neighbor nodes.
3. Construct the out-edge of each point in the selected `candidate_pool_size` nodes according to the edge selection strategy.

The query process is similar to the graph building process. It starts from the navigation point and iterates at least `search_length` times to get the final result.

- **building parameters:**
  - `search_length`: Number of query iterations.
  - `out_degree`: Maximum out-degree of the node.
  - `candidate_pool_size`: Candidate pool size of the node.
  - `knng`: Number of nearest neighbors

```python
# RNSG
client.create_index(collection_name,
    IndexType.RNSG,
    {
        "search_length": 60,       # int. 10~300
        "out_degree": 30,          # int. 5~300
        "candidate_pool_size": 300, # int. 50~1000
        "knng": 50                # int. 5~300
    }
)
```

- **search parameters:**
  - `search_length`: Number of query iterations

```python
# RNSG
client.search(collection_name,
    1,
    query_vectors,
    params={
        "search_length": 100  # int. 10~300
    }
)
```

*Section author: Godchen@milvus*
1.5 Search results

1.5.1 How to deal with search results

The invocation of `search()` is like this:

```python
>>> results = client.search('demo', query_vectors, topk)
```

The result object can be used as a 2-D array. `results[i] (0 <= i < len(results))` represents topk results of i-th query vector, and `results[i][j] (0 <= j < len(results[i]))` represents j-th result of i-th query vector. To get result id and distance, you can invoke like this:

```python
>>> id = results[i][j].id
>>> distance = results[i][j].distance
```

The results object can be iterated, so you can traverse the results with two-level loop:

```python
>>> for raw_result in results:
...     for result in raw_result:
...         id = result.id  # result id
...         distance = result.distance
```

Meanwhile, the results object provide attributes to separately access result id array `id_array` and distance array `distance_array`, so you can traverse the results like this:

```python
>>> for ids, distances in zip(results.id_array, results.distance_array):
...     for id_, dis_ in zip(ids, distances):
...         print(f"id = {id_}, distance = {dis_}")
```

Section author: Bosszou@milvus

1.6 Changelog

v1.0.2(Developing)

v1.0.1

- Remove unused hybrid APIs

Section author: Bosszou@milvus

1.7 FAQ

- *I'm getting random "socket operation on non-socket" errors from gRPC.*
- *How to fix the error when I install PyMilvus on Windows?*
1.7.1 I’m getting random “socket operation on non-socket” errors from gRPC.

Make sure to set the environment variable GRPC_ENABLE_FORK_SUPPORT=1. For reference, see this post.

1.7.2 How to fix the error when I install PyMilvus on Windows?

Try installing PyMilvus in a Conda environment.

Section author: Yangxuan@milvus

1.8 Contributing

- Open Issues
- Submit Pull Requests
- Github workflow
- Contribution Guideline

Contributing is warmly welcomed. You can contribute to PyMilvus project by opening issues and submitting pull requests on PyMilvus Github page.

1.8.1 Open Issues

To request a new feature, report a bug or ask a question, it’s recommended for you to open an issue.

For a feature You can tell us why you need it and we will decide whether to implement it soon. If we think it’s a good improvement, we will make it a feature request and start to work on it. It’s also welcomed for you to open an issue with your PR as a solution.

For a bug You need to tell us as much information as possible, better start with our bug report template. With information, we can reproduce the bug easily and solve it later.

For a question It’s welcomed to ask any questions about PyMilvus and Milvus, we are pleased to communicate with you.

1.8.2 Submit Pull Requests

If you have improvements to PyMilvus, please submit pull requests(PR) to master, see workflow below.

PR for codes, you need to tell us why we need it, mentioning an existing issue would be better.

PR for docs, you also need to tell us why we need it.

Your PRs will be reviewed and checked, merged into our project if approved.
1.8.3 Github workflow

This is a brief instruction of Github workflow for beginners.

- **Fork** the PyMilvus repository on Github.
- **Clone** your fork to your local machine with `git clone git@github.com:<your_user_name>/pymilvus.git`.
- Create a new branch with `git checkout -b my_working_branch`.
- Make your changes, commit, then push to your forked repository.
- Visit Github and make you PR.

If you already have an existing local repository, always update it before you start to make changes like below:

```
$ git remote add upstream git@github.com:milvus-io/pymilvus.git
$ git checkout master
$ git pull upstream master
$ git checkout -b my_working_branch
```

1.8.4 Contribution guideline

1. Update CHANGELOG.md

If any improvement or feature being added, you are recommended to open a new issue(if not exist) then record your change in file `CHANGELOG.md`. The format is: `- #{GitHub issue number} - {Brief description for your change}`

2. Add unit tests for your codes

To run unit test in github action, you need make sure the last commit message of PR starts with “[ci]”. If you want to run unit test locally, under root folder of Pymilvus project run `pytest --ip=${IP} --port=${PORT}`.

3. Pass pylint check

In the root directory, run `pylint --rcfile=pylint.conf milvus/client` to make sure the rate is 10.

4. For documentations

You need to enter the doc directory and run `make html`, please refer to About this documentations.

*Section author: Yangxuan@milvus*

1.9 About this documentation

This documentation is generated using the Sphinx documentation generator. The source files for the documentation are located in the `doc/` directory of the pymilvus distribution. To generate the docs locally run the following command under directory `doc/`:

```
$ make html
```

The documentation should be generated under directory `build/html`.

To preview it, you can open `index.html` in your browser.

Or run a web server in that directory:

```
$ python3 -m http.server
```
Then open your browser to http://localhost:8000.

*Section author:* Bosszou@milvus

*Section author:* Bosszou@milvus, Godchen@milvus, Yangxuan@milvus
INDEX

A
ANNOY (milvus.IndexType attribute), 15

C
compact() (milvus.Milvus method), 14
count_entities() (milvus.Milvus method), 9
create_collection() (milvus.Milvus method), 8
create_index() (milvus.Milvus method), 11
create_partition() (milvus.Milvus method), 12

D
delete_entity_by_id() (milvus.Milvus method), 14
drop_collection() (milvus.Milvus method), 10
drop_index() (milvus.Milvus method), 12
drop_partition() (milvus.Milvus method), 13

F
FLAT (milvus.IndexType attribute), 15
flush() (milvus.Milvus method), 14

G
get_collection_info() (milvus.Milvus method), 9
get_collection_stats() (milvus.Milvus method), 9
get_entity_by_id() (milvus.Milvus method), 11
get_index_info() (milvus.Milvus method), 12

H
HAMMING (milvus.MetricType attribute), 16
has_collection() (milvus.Milvus method), 9
has_partition() (milvus.Milvus method), 12
HNSW (milvus.IndexType attribute), 15

I
IndexType (class in milvus), 15
insert() (milvus.Milvus method), 10
INVALID (milvus.IndexType attribute), 15
INVALID (milvus.MetricType attribute), 15
IP (milvus.MetricType attribute), 15
IVF_FLAT (milvus.IndexType attribute), 15
IVF_PQ (milvus.IndexType attribute), 15
IVF_SQ8 (milvus.IndexType attribute), 15
IVF_SQ8_H (milvus.IndexType attribute), 15
IVF_SQ8H (milvus.IndexType attribute), 15
IVFLAT (milvus.IndexType attribute), 15
J
JACCARD (milvus.MetricType attribute), 16

L
L2 (milvus.MetricType attribute), 15
list_collections() (milvus.Milvus method), 9
list_id_in_segment() (milvus.Milvus method), 11
list_partitions() (milvus.Milvus method), 13
load_collection() (milvus.Milvus method), 10

M
MetricType (class in milvus), 15
Milvus (class in milvus), 8

R
release_collection() (milvus.Milvus method), 10
RNSG (milvus.IndexType attribute), 15

S
search() (milvus.Milvus method), 13
SUBSTRUCTURE (milvus.MetricType attribute), 16
SUPERSTRUCTURE (milvus.MetricType attribute), 16

T
TANIMOTO (milvus.MetricType attribute), 16