

OCEANBASE



OceanBase 数据库

向量检索

| 产品版本：V4.3.5


| 文档版本：20250106

声明

北京奥星贝斯科技有限公司版权所有©2024，并保留一切权利。

未经北京奥星贝斯科技有限公司事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

商标声明

 **OCEANBASE** 及其他 OceanBase 相关的商标均为北京奥星贝斯科技有限公司所有。本文档涉及的第三方的注册商标，依法由权利人所有。

免责声明

由于产品版本升级、调整或其他原因，本文档内容有可能变更。北京奥星贝斯科技有限公司保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在北京奥星贝斯科技有限公司授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过北京奥星贝斯科技有限公司授权渠道下载、获取最新版的用户文档。如因文档使用不当造成的直接或间接损失，本公司不承担任何责任。

通用约定

格式	说明	样例
 危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	 危险 重置操作将丢失用户配置数据。
 警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	 警告 重启操作将导致业务中断，恢复业务时间约十分钟。
 注意	用于警示信息、补充说明等，是用户必须了解的内容。	 注意 权重设置为0，该服务器不会再接受新请求。
 说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	 说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击 设置> 网络> 设置网络类型 。
粗体	表示按键、菜单、页面名称等UI元素。	在 结果确认 页面，单击 确定 。
Courier字体	命令或代码。	执行 <code>cd /d C:/window</code> 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	<code>bae log list --instanceid</code> <code>Instance_ID</code>
[] 或者 [a b]	表示可选项，至多选择一个。	<code>ipconfig [-all -t]</code>
{ } 或者 {a b}	表示必选项，至多选择一个。	<code>switch {active stand}</code>

目录

1 向量检索概述	8
1.1 关键概念	8
1.1.1 非结构化数据	8
1.1.2 向量	8
1.1.3 向量嵌入(Embedding)	8
1.1.4 向量相似性检索	8
1.2 为什么选择 OceanBase 向量检索?	9
1.2.1 多模融合查询	9
1.2.2 分布式可扩展	9
1.2.3 高性能	9
1.2.4 高可用	9
1.2.5 事务性	9
1.2.6 低成本	9
1.2.7 数据安全	9
1.2.8 简单易用	9
1.2.9 完备的工具体系	10
1.3 应用场景	10
2 使用 SQL 快速进行向量检索	11
2.1 前提条件	11
2.2 快速上手	11
2.2.1 启用向量检索功能	11
2.2.2 创建向量列和索引	11
2.2.3 写入向量数据	12
2.2.4 执行向量检索	13
2.3 相关文档	13

3	使用 Python 快速进行向量检索	14
3.1	前提条件	14
3.2	快速上手	14
3.2.1	安装 pyobvector	14
3.2.2	用法	14
3.2.3	Milvus 兼容模式	14
3.2.3.1	建立客户端连接	14
3.2.3.2	创建具有向量索引的集合	15
3.2.3.3	构造并写入数据	16
3.2.3.4	执行相似向量检索	16
3.2.4	SQLAlchemy 混合模式	17
3.2.4.1	建立客户端连接	17
3.2.4.2	创建表和向量索引	17
3.2.4.3	构造并写入数据	18
3.2.4.4	执行相似向量检索	19
3.3	相关文档	20
4	向量检索核心功能	21
4.1	使用限制	21
5	向量检索文本嵌入	22
5.1	什么是文本嵌入?	22
5.2	常见的文本嵌入方法	22
5.2.1	操作准备	22
5.2.2	使用离线、本地的预训练嵌入模型	22
5.2.2.1	使用 Sentence Transformers	22
5.2.2.2	使用 Hugging Face Transformers	23
5.2.2.3	Ollama	24
5.2.3	使用在线、远端的嵌入服务	26
5.2.3.1	HTTP 调用	28
5.2.3.2	使用通义千问 SDK	31
6	向量数据库迁移指南	33
6.1	Milvus 数据迁移	33
6.2	Qdrant 数据迁移	33
7	向量检索实践教程概述	34
7.1	向量检索实践教程	34
7.1.1	基于 OceanBase 构建智能问答机器人	34
7.1.2	基于 OceanBase 构建图搜图应用	34
8	AI 集成概述	35
8.1	AI 集成应用场景	35
8.1.1	OpenAI API 集成	35
8.1.2	通义千问 API 集成	35
8.1.3	Langchain 集成	35
8.1.4	LlamaIndex 集成	36

9 向量数据类型概述	37
9.1 语法	37
10 向量函数	38
10.1 使用注意事项	38
10.2 距离函数	38
10.2.1 L2_distance	38
10.2.2 L1_distance	39
10.2.3 Cosine_distance	39
10.2.4 Inner_product	40
10.2.5 Vector_distance	41
10.3 算术函数	42
10.4 比较函数	43
10.4.0.1 注意	43
10.5 聚合函数	44
10.5.0.1 注意	44
10.5.1 Sum	44
10.5.2 Avg	44
10.6 其他常见向量函数	45
10.6.1 Vector_norm	45
10.6.2 Vector_dims	45
11 向量索引	47
11.0.0.1 注意	47
11.1 索引查询概述	47
11.2 索引创建、查询及删除示例	48
11.2.1 注意事项	48
11.2.2 建表时创建索引	48
11.2.3 后建索引	50
11.2.4 删除索引	52
11.3 索引维护	52
11.3.1 增量刷新	52
11.3.2 全量刷新	53
11.3.2.1 索引维护示例	53
11.4 相关文档	54
12 兼容性说明	55
12.1 与 Milvus 概念映射	55
12.1.1 数据模型	55
12.1.2 SDK	55
12.2 与 Milvus 兼容性	57
12.2.1 Milvus SDK	57
12.2.2 pymilvus	59
12.2.2.1 数据模型	59
12.2.2.2 Milvus Lite API 兼容性	59
12.3 与 MySQL 协议兼容性	65

13	pyobvector Python SDK 接口说明	66
13.1	MilvusLikeClient	66
13.1.1	构造函数	66
13.1.2	collection 相关接口	66
13.1.3	CollectionSchema FieldSchema	69
13.1.3.1	通过 MilvusLikeClient 的 create_schema 创建 CollectionSchema	
13.1.3.2	创建 FieldSchema 并注册到 CollectionSchema	70 69
13.1.3.3	使用示例	70
13.1.4	索引相关	71
13.2	ObVecClient	77
13.2.1	构造函数	77
13.2.2	表模式相关操作	77
13.2.3	DML 操作	81
13.3	使用 ObPartition 定义分区规则	86
13.3.1	range 分区示例	87
13.3.2	list 分区示例	87
13.3.3	hash 分区示例	87
13.3.4	多级分区示例	88
13.4	纯 SQLAlchemy API 模式	88
13.5	更多示例	90
14	向量检索常见问题	91
14.1	在什么模式下支持向量检索?	91
14.2	向量列中, 每一行的数据维度是否必须相同?	91
14.3	最大支持写入多少行的向量数据?	91
14.4	如何对超过 2000 维的向量建索引?	91
14.5	OceanBase 是否会支持内置的 Embedding 方法和内置的模型算法? ...	91

1 向量检索概述

本文档介绍了向量数据库与向量检索的核心概念。

OceanBase 数据库最高支持 16000 维的 Float 类型的稠密向量，支持曼哈顿距离、欧式距离、内积、余弦距离等多种类型向量距离的计算，支持基于 HNSW 向量索引的创建，支持增量更新删除，同时增量更新删除操作召回率不会产生影响。OceanBase 向量检索具备带有标量过滤的融合查询能力。同时提供灵活的访问接口，不仅支持通过 MySQL 协议各种语言客户端使用 SQL 访问，也可以使用 Python SDK 访问。同时 OceanBase 数据库也完成了对 AI 应用开发框架 LlamaIndex、DB-GPT 及 AI 应用开发平台 Dify 的适配，更好的服务于 AI 应用开发。

1.1 关键概念

1.1.1 非结构化数据

非结构化数据是指没有明确定义的数据格式和组织结构的数据。非结构化数据通常包括文本、图像、音频、视频等形式的数据，以及社交媒体内容、电子邮件、日志文件等。由于非结构化数据的复杂性和多样性，处理这些数据需要采用特定的工具和技术，例如自然语言处理、图像识别、机器学习等。

1.1.2 向量

向量本质上是一个对象在高维空间的投影。数学意义上向量则是一个浮点数组，有以下两个特点：

- 数组中每个元素表示向量的某个维度，每个元素都是一个浮点数。
- 向量数组的大小（元素个数）表示整个向量空间的维度。

1.1.3 向量嵌入(Embedding)

向量嵌入(Embedding) 指的是通过深度学习神经网络提取非结构化数据里的内容和语义，把图片、视频等变成特征向量的过程。Embedding 技术将原始数据从高维度(稀疏)空间映射到低维度(稠密)空间，将具有丰富特征的多模态数据转换为多维数组(向量)。

1.1.4 向量相似性检索

在当今信息爆炸的时代，用户常需要从海量数据中迅速检索所需信息。例如在线文献数据库、电商平台产品目录、以及不断增长的多媒体内容库，都需要高效的检索系统来快速定位到用户感兴趣的内容。随着数据量不断激增，传统的基于关键字的检索方法已经无法满足用户对于检索精度和速度的需求，向量检索技术应运而生。向量相似性检索使用特征提取和向量化技术将文本、图片、音频等不同类型的非结构化数据转换为向量，使用相似性度量方法来比较它们之间的相似性，进而捕捉数据的深层次语义信息，从而提供更为准确和高效的检索结果。

1.2 为什么选择 OceanBase 向量检索？

OceanBase 向量检索能力基于 OceanBase 多模一体化能力上构建，在融合查询、扩展性、高性能、高可用、低成本、多租户、数据安全等方面均有优异的表现。

1.2.5 多模融合查询

OceanBase 支持向量数据、空间数据、文档数据、标量数据等类型融合查询，基于向量索引、空间索引、全文索引的等多种索引的支持，提供极致性能的多模融合查询能力。OceanBase 真正实现用一套数据库解决应用多样存储检索需求。

1.2.6 分布式可扩展

OceanBase 作为原生分布式数据库，其水平扩展能力及多分区能力，支撑 OceanBase 对海量向量数据的支持。

1.2.7 高性能

OceanBase 向量检索能力集成了索引算法库 VSAG，VSAG 算法库在 960 维的 GIST 数据集上表现出色，在 ANN-Benchmarks 测试中远超其他算法。

1.2.8 高可用

基于 Paxos+ 数据同步容灾方案，OceanBase 向量检索也支持主备/跨机房/跨地域容灾，对于基于内存的 HNSW 索引，容灾切换后也能实时访问。

1.2.9 事务性

OceanBase 数据库基于 Multi-Paxos 协议的分布式事务能力不仅保证了向量数据的一致性和完整性，还提供了有效的并发控制和故障恢复机制。

1.2.10 低成本

OceanBase 的存储编码压缩能力能够显著降低向量存储空间，节省应用的存储成本。

1.2.11 数据安全

OceanBase 数据库已经支持比较完整的企业级安全特性，包括身份鉴别和认证、访问控制、数据加密、监控告警、安全审计，可以有效保证向量检索场景下的数据安全。

1.2.12 简单易用

OceanBase 向量检索提供灵活的访问接口，不仅支持通过 MySQL 协议各种语言客户端使用 SQL 访问，也可以使用 Python SDK 访问。同时 OceanBase 也完成了对 AI 应用开发框架 LangChain 和 Llamaindex 的适配，更好的服务于 AI 应用开发。

1.2.13 完备的工具体系

OceanBase 具备完备的数据库工具体系，支持数据开发、迁移、运维、诊断等数据全生命周期的管理，给 AI 应用的开发维护保驾护航。

1.3 应用场景

- RAG（Retrieval Augmented Generation）检索增强生成：RAG 是一个人工智能框架，用于从外部知识库中检索事实，以便为大型语言模型 (LLM) 提供最准确、最新的信息，并让用户深入了解 LLM 的生成过程，常应用于智能问答、知识库等。
- 个性化推荐：推荐系统可以根据用户的历史行为和偏好，向用户推荐可能感兴趣的物品。当发起推荐请求时，系统会基于用户特征进行相似度计算，然后返回与用户可能感兴趣的物品作为推荐结果，如饭店推荐、景点推荐等。
- 图搜图/文本搜图：图像/文本检索任务是指在大规模图像/文本数据库中搜索出与指定图像最相似的结果，在检索时使用到的文本/图像特征可以存储在向量数据库中，通过高性能的索引存储实现高效的相似度计算，进而返回和检索内容相匹配的图像/文本结果，如人脸识别等。

2 使用 SQL 快速进行向量检索

OceanBase 支持使用 SQL 语言进行向量检索，本文介绍了快速上手方法。

2.1 前提条件

- 请确保你已[部署了 OceanBase 集群](#)并创建了 MySQL 租户。
- 已连接到 OceanBase 数据库。详情请参考[连接方式概述](#)。

2.2 快速上手

2.2.1 启用向量检索功能

在使用向量索引前，需要依据租户下的索引数据估计内存占用并进行配置，内存计算方式请参见 [ob_vector_memory_limit_percentage](#)。以下命令表示将向量索引可用内存配置为租户内存的 30%：

```
ALTER SYSTEM SET ob_vector_memory_limit_percentage = 30;
```

`ob_vector_memory_limit_percentage` 的默认值为 `0`，表示不为向量索引分配内存，创建索引会报错。

2.2.2 创建向量列和索引

创建表时，可以使用 `VECTOR(dim)` 数据类型声明指定列为向量列及其维度。向量索引需要创建在向量列上，且至少需要提供 `type` 和 `distance` 两个参数。

示例中创建向量列 `embedding`，向量数据维度为 `3`，并在 `embedding` 列上创建 HNSW 索引，指定距离算法为 L2。

```
CREATE TABLE t1(  
id INT PRIMARY KEY,  
doc VARCHAR(200),  
embedding VECTOR(3),  
VECTOR INDEX idx1(embedding) WITH (distance=L2, type=hnsw)  
);
```

在数据量较大的情况下建议先导入完数据再创建向量索引，后建索引详情请参见[向量检索索引](#)。

2.2.3 写入向量数据

为了模拟在向量检索的场景，需要先构造一些向量数据，每行数据都对数据的描述和对应的向量。示例中假设 '苹果' 对应的向量为 '[1.2,0.7,1.1]'，'胡萝卜' 对应的向量为 '[5.3,4.8,5.4]' 等。

```
INSERT INTO t1
VALUES (1, '苹果', '[1.2,0.7,1.1]'),
(2, '香蕉', '[0.6,1.2,0.8]'),
(3, '橙子', '[1.1,1.1,0.9]'),
(4, '胡萝卜', '[5.3,4.8,5.4]'),
(5, '菠菜', '[4.9,5.3,4.8]'),
(6, '西红柿', '[5.2,4.9,5.1]);
```

为了方便展示，本例简化了向量的维度，仅使用了 3 维向量，且向量是人工生成的。在实际应用中，需要使用嵌入模型对真实的文本进行生成，维度会达到数百或上千维。

可以通过查询表中的数据查看是否写入成功。

```
SELECT * FROM t1;
```

预期返回结果如下：

```
+-----+-----+-----+
| id | doc | embedding |
+-----+-----+-----+
| 1 | 苹果 | [1.2,0.7,1.1] |
| 2 | 香蕉 | [0.6,1.2,0.8] |
| 3 | 橙子 | [1.1,1.1,0.9] |
| 4 | 胡萝卜 | [5.3,4.8,5.4] |
| 5 | 菠菜 | [4.9,5.3,4.8] |
| 6 | 西红柿 | [5.2,4.9,5.1] |
+-----+-----+-----+
6 rows in set
```

2.2.4 执行向量检索

进行向量检索需要提供向量作为搜索条件。假设我们需要找到所有 '水果'，其对应的向量为 [0.9, 1.0, 0.9]，则对应 SQL 为：

```
SELECT id, doc FROM t1
ORDER BY l2_distance(embedding, '[0.9, 1.0, 0.9]')
APPROXIMATE LIMIT 3;
```

预期返回结果如下：

```
+-----+-----+
| id | doc |
+-----+-----+
| 3 | 橙子 |
| 2 | 香蕉 |
| 1 | 苹果 |
+-----+-----+
3 rows in set
```

2.3 相关文档

- 向量数据详细说明请参见 [向量数据](#)。
- 建表后创建向量索引、删除索引方法请参见 [向量索引](#)。

3 使用 Python 快速进行向量检索

OceanBase 支持使用 `pyobvector` 进行向量存储和检索，本文介绍了快速上手方法。`pyobvector` 是 OceanBase 向量存储的 Python SDK，基于 `SQLAlchemy`，基本兼容 `Milvus API`。

3.1 前提条件

- 请确保你已[部署了 OceanBase 集群](#)并创建了 MySQL 租户。
- 请确保你的环境已安装 Python 3.9 及以上版本。

3.2 快速上手

3.2.1 安装 `pyobvector`

首先，需要将 `pyobvector` 先安装到我们的本地环境中。可参考如下命令：

```
pip install -U pyobvector
```

`-U` 参数表示如果你的环境已经安装了 `pyobvector`，则自动升级到最新版本；如果未安装，则直接安装最新版本。

3.2.2 用法

`pyobvector` 支持两种模式：

- `Milvus` 兼容模式：用 `MilvusLikeClient` 类提供的类似于 `Milvus API` 的方式来使用向量存储。
- `SQLAlchemy` 混合模式：使用 `ObVecClient` 类提供的向量存储功能，并用 `SQLAlchemy` 库执行关系数据库语句。在这种模式下，可以将 `pyobvector` 视为 `SQLAlchemy` 的扩展。

3.2.3 `Milvus` 兼容模式

3.2.3.1 建立客户端连接

`pyobvector` 提供了 `MilvusLikeClient` 让用户能够以 `Milvus` 兼容的方式使用 OceanBase 的向量存储和检索能力，通过下面的语句能够创建一个客户端对象：

```
from pyobvector import *
```

请将下面的数据库连接信息参数修改为您的数据库实例的信息

```
client = MilvusLikeClient(uri="127.0.0.1:2881", user="root@test", db_name="test")
```

3.2.3.2 创建具有向量索引的集合

为了兼容 Milvus 的 API, pyobvector 的 MilvusClient 也提供了 `create_collection` 等方法。虽然方法名是创建集合, 但是映射到 OceanBase 中实际上是创建了一张表, 通过下面的例子能够创建一张具有 `id`、`embedding`、`metadata` 三列的表, 并且针对 `embedding` 列创建一个 HNSW 类型的向量索引。其中 `embedding` 列就是一个 64 维的向量类型。

```
fields = [
    FieldSchema(
        name="id",
        dtype=DataType.INT64,
        is_primary=True,
        auto_id=True,
    ),
    FieldSchema(name="embedding", dtype=DataType.FLOAT_VECTOR, dim=64),
    FieldSchema(name="metadata", dtype=DataType.JSON),
]

index_params = MilvusLikeClient.prepare_index_params()
index_params.add_index(
    field_name="embedding",
    index_name="embedding_idx",
    index_type=VecIndexType.HNSW,
    distance="l2",
    m=16,
    ef_construction=256,
)
```

```
schema = CollectionSchema(fields)
table_name = "vector_search"
client.create_collection(table_name, schema=schema, index_params=index_params)
```

3.2.3.3 构造并写入数据

为了模拟在大量向量数据中进行检索的场景，在这一步中先构造一些向量数据。方式是使用 python 中的 random 模块来构造随机的浮点数列表。

```
import random

random.seed(20241023)

batch_size = 100
batch = []
for i in range(1000):
    batch.append(
        {
            "embedding": [random.uniform(-1, 1) for _ in range(64)],
            "metadata": {"idx": i},
        }
    )
    if len(batch) == batch_size:
        client.insert(collection_name=table_name, data=batch)
        batch = []

if len(batch) > 0:
    client.insert(collection_name=table_name, data=batch)
```

3.2.3.4 执行相似向量检索

通过 random.uniform 构造一个目标向量数据作为输入，在刚才插入数据的集合中进行向量检索：


```
target_data = [random.uniform(-1, 1) for _ in range(64)]
res = client.search(
    collection_name=table_name,
    data=target_data,
    anns_field="embedding",
    limit=5,
    output_fields=["id", "metadata"],
)
print(res)
# 预期返回结果如下
# [{'id': 63, 'metadata': {'idx': 62}}, {'id': 796, 'metadata': {'idx': 795}}, {'id': 187,
'metadata': {'idx': 186}}, {'id': 784, 'metadata': {'idx': 783}}, {'id': 880, 'metadata':
{'idx': 879}}]
```

3.2.4 SQLAlchemy 混合模式

3.2.4.5 建立客户端连接

pyobvector 提供了 ObVecClient 让用户能够以 SQLAlchemy 混合模式使用 OceanBase 的向量存储和检索能力，通过下面的语句能够创建一个客户端对象：

```
from pyobvector import *

# 请将下面的数据库连接信息参数修改为您的数据库实例的信息
client = ObVecClient(uri="127.0.0.1:2881", user="root@test", db_name="test")
```

3.2.4.6 创建表和向量索引

通过下面的例子能够创建一张具有 `id`、`embedding`、`metadata` 三列的表，并且针对 `embedding` 列创建一个 HNSW 类型的向量索引。其中 `embedding` 列就是一个 64 维的向量类型。

```
from sqlalchemy import Column, Integer, JSON
from sqlalchemy import func

cols = [
    Column("id", Integer, primary_key=True, autoincrement=True),
    Column("embedding", VECTOR(64)),
    Column("metadata", JSON),
]
table_name = "vector_test3"
client.create_table(table_name, columns=cols)
print(f"Table {table_name} created")
client.create_index(
    table_name,
    is_vec_index=True,
    index_name="embedding_idx",
    column_names=["embedding"],
    vidx_params="distance=l2, type=hns, lib=vsag", # m=16, ef_construction=256
)
print(f"Index {table_name}.embedding_idx created")
```

3.2.4.7 构造并写入数据

为了模拟在大量向量数据中进行检索的场景，在这一步中先构造一些向量数据。方式是使用python 中的 random 模块来构造随机的浮点数列表。

```
import random

random.seed(20241023)

batch_size = 100
batch = []
```

```
for i in range(1000):
    batch.append(
        {
            "embedding": [random.uniform(-1, 1) for _ in range(64)],
            "metadata": {"idx": i},
        }
    )
    if len(batch) == batch_size:
        client.insert(table_name, data=batch)
        batch = []

    if len(batch) > 0:
        client.insert(table_name, data=batch)
```

3.2.4.8 执行相似向量检索

通过 `random.uniform` 构造一个目标向量数据作为输入，在刚才插入数据的集合中进行向量检索：

```
target_data = [random.uniform(-1, 1) for _ in range(64)]
res = client.ann_search(
    table_name,
    vec_data=target_data,
    vec_column_name="embedding",
    distance_func=func.l2_distance,
    topk=5,
    output_column_names=["id", "metadata"],
)
for r in res:
    print(r)
# 预期返回结果如下
```

```
# (63, '{"idx": 62}')  
# (796, '{"idx": 795}')  
# (187, '{"idx": 186}')  
# (784, '{"idx": 783}')  
# (880, '{"idx": 879}')
```

3.3 相关文档

- 向量数据详细说明请参见 [向量数据](#)。
- 建表后创建向量索引、删除索引方法请参见 [向量索引](#)。

4 向量检索核心功能

OceanBase 提供了存储、索引、检索 Embedding 向量数据的能力。具体包括：

核心功能	描述
向量数据类型	支持最大 16,000 维的 float 向量数据存储。
向量索引	<ul style="list-style-type: none">支持精确搜索和近似最近邻搜索。支持 L2 距离、内积和余弦相似度计算。支持 HNSW 索引，索引列最大维度为 4096。
向量搜索 SQL 运算符	支持向量加、减、乘、比较、聚合等基础运算操作符。

4.1 使用限制

- OceanBase 默认采用 NULL first 比较模式，所以对 NULL 值进行排序时会将其放至最前，建议查询的时候加上 NOT NULL 条件。
- 暂不支持在一张表上同时定义向量索引和全文索引。

5 向量检索文本嵌入

本文档介绍了向量检索中的文本嵌入概念并给出了一些示例。

5.1 什么是文本嵌入？

文本嵌入是一种将文本转换为数值向量的技术。这些向量能够捕捉文本的语义信息，使计算机可以“理解”和处理文本的含义。具体来说：

- 文本嵌入将词语或句子映射到高维向量空间中的点。
- 在这个向量空间中，语义相似的文本会被映射到相近的位置。
- 向量通常由数百个数字组成(如 512 维、1024 维等)。
- 可以用数学方法(如余弦相似度)计算向量之间的相似度。
- 常见的文本嵌入模型包括 Word2Vec、BERT、BGE 等。

在开发 RAG 应用时，我们通常需要将文本数据进行嵌入处理转换为向量之后存储在向量数据库中，而其他结构化数据存储在关系型数据库中。OceanBase 4.3.3 版本开始支持将向量作为一种字段类型在关系表中进行存储，使得向量和传统标量数据能够有序、高效地存储在 OceanBase 这一款数据库中。

5.2 常见的文本嵌入方法

5.2.1 操作准备

您需要提前安装好 `pip` 命令。

5.2.2 使用离线、本地的预训练嵌入模型

使用预训练模型在本地进行文本嵌入是最灵活的方式，但需要较大的计算资源。常用的模型包括：

5.2.2.1 使用 Sentence Transformers

因为在国内直接访问 `hugging face` 的域名通常会超时，请提前设置 `hugging face` 的镜像地址 `export HF_ENDPOINT=https://hf-mirror.com`，设置完成后再执行下面的代码：

```
from sentence_transformers import SentenceTransformer

model = SentenceTransformer("BAAI/bge-m3")
```

```
sentences = [
    "That is a happy person",
    "That is a happy dog",
    "That is a very happy person",
    "Today is a sunny day"
]
embeddings = model.encode(sentences)
print(embeddings)
# [[-0.01178016 0.00884024 -0.05844684 ... 0.00750248 -0.04790139
# 0.00330675]
# [-0.03470375 -0.00886354 -0.05242309 ... 0.00899352 -0.02396279
# 0.02985837]
# [-0.01356584 0.01900942 -0.05800966 ... 0.00523864 -0.05689549
# 0.00077098]
# [-0.02149693 0.02998871 -0.05638731 ... 0.01443702 -0.02131325
# -0.00112451]]
similarities = model.similarity(embeddings, embeddings)
print(similarities.shape)
# torch.Size([4, 4])
```

5.2.2.2 使用 Hugging Face Transformers

因为在国内直接访问 hugging face 的域名通常会超时，请提前设置 hugging face 的镜像地址 `export HF_ENDPOINT=https://hf-mirror.com`，设置完成后再执行下面的代码：

```
from transformers import AutoTokenizer, AutoModel
import torch

# 加载模型和分词器
tokenizer = AutoTokenizer.from_pretrained("BAAI/bge-m3")
model = AutoModel.from_pretrained("BAAI/bge-m3")
```

```
# 准备输入
texts = ["这是示例文本"]
inputs = tokenizer(texts, padding=True, truncation=True, return_tensors="pt")

# 生成嵌入
with torch.no_grad():
    outputs = model(**inputs)
    embeddings = outputs.last_hidden_state[:, 0] # 使用 [CLS] token 的输出
    print(embeddings)
# tensor([[[-1.4136, 0.7477, -0.9914, ..., 0.0937, -0.0362, -0.1650]])
print(embeddings.shape)
# torch.Size([1, 1024])
```

5.2.2.3 Ollama

[Ollama](#) 是一个开源的模型运行时，它让用户能够在本地轻松运行、管理和使用各种大语言模型。除了支持 Llama 3 和 Mistral 等开源语言模型外，它还支持 bge-m3 等嵌入模型。

1. 部署 Ollama

在 MacOS 和 Windows 上可以直接从官网下载安装包进行安装，安装方法可参考 Ollama 的官网。安装完成后，Ollama 会作为一个服务在后台运行。

在 Linux 上安装 Ollama：

```
curl -fsSL https://ollama.ai/install.sh | sh
```

2. 拉取嵌入模型

Ollama 支持使用 bge-m3 模型用于文本嵌入：

```
ollama pull bge-m3
```

3. 使用 Ollama 进行文本嵌入

可以通过 HTTP API 或者 Python SDK 等方式来使用 Ollama 的嵌入能力：

- HTTP API 方式


```
import requests

def get_embedding(text: str) -> list:
    """使用 Ollama 的 HTTP API 获取文本嵌入"""
    response = requests.post(
        'http://localhost:11434/api/embeddings',
        json={
            'model': 'bge-m3',
            'prompt': text
        }
    )
    return response.json()['embedding']

# 示例使用
text = "这是一个示例文本"
embedding = get_embedding(text)
print(embedding)
# [-1.4269912242889404, 0.9092104434967041, ...]
```

- Python SDK 方式

首先安装 Ollama 的 Python SDK:

```
pip install ollama
```

然后可以这样使用:

```
import ollama

# 示例使用
texts = ["第一个句子", "第二个句子"]
embeddings = ollama.embed(model="bge-m3", input=texts)['embeddings']
```

```
print(embeddings)
# [[0.03486196, 0.0625187, ...], [...]]
```

4. Ollama 的优势和局限

优势：

- 完全本地部署，无需网络连接
- 开源免费，无需 API Key
- 支持多种模型，便于切换和比较
- 资源占用相对较小

局限：

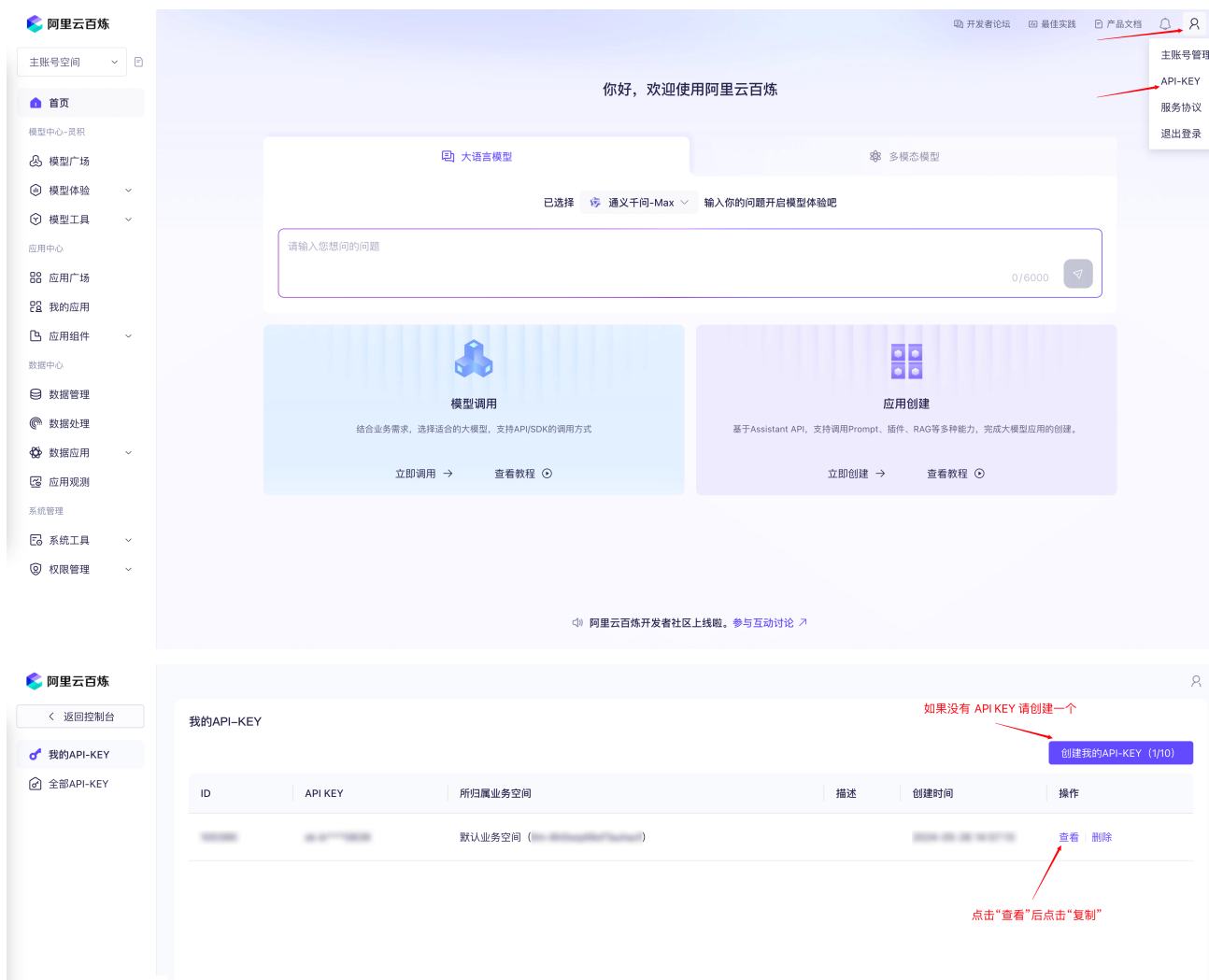
- 嵌入模型选择较少
- 性能可能不如商业服务
- 需要自行维护和更新
- 缺乏企业级支持

在选择是否使用 Ollama 时，需要权衡这些因素。如果您的应用场景对隐私性要求高，或者希望完全离线运行，Ollama 是一个不错的选择。但如果您需要更稳定的服务质量和更好的性能，可能还是需要考虑商业服务。

5.2.3 使用在线、远端的嵌入服务

使用离线的、本地的嵌入模型通常对部署的机器规格要求较高，而且对模型的加载和卸载等过程的管理有较高的要求，许多用户对在线的嵌入服务有较高需求，所以目前许多 AI 推理服务提供商也提供了相应的文本嵌入服务。以通义千问的文本嵌入服务为例，我们首先注册[阿里云百炼](#)账号并获取 API Key 后，便可以通过调用其公开的接口来得到文本嵌入的结果。





5.2.3.4 HTTP 调用

获取完成之后可通过下面的代码尝试进行文本嵌入，如果您的 Python 环境中没有安装 requests 包，则需要先通过 `pip install requests` 进行安装以便发送网络请求。

```
import requests
from typing import List

class RemoteEmbedding():
    def __init__(
        self,
        base_url: str,
        api_key: str,
        model: str,
```

```
dimensions: int = 1024,
**kwargs,
):
self._base_url = base_url
self._api_key = api_key
self._model = model
self._dimensions = dimensions

"""
OpenAI compatible embedding API. Tongyi, Baichuan, Doubao, etc.
"""

def embed_documents(
self,
texts: List[str],
) -> List[List[float]]:
"""Embed search docs.

Args:
texts: List of text to embed.

Returns:
List of embeddings.
"""
res = requests.post(
f"{self._base_url}",
headers={"Authorization": f"Bearer {self._api_key}"},
json={
"input": texts,
```

```
"model": self._model,
"encoding_format": "float",
"dimensions": self._dimensions,
},
)
data = res.json()
embeddings = []
try:
for d in data["data"]:
embeddings.append(d["embedding"][: self._dimensions])
return embeddings
except Exception as e:
print(data)
print("Error", e)
raise e

def embed_query(self, text: str, **kwargs) -> List[float]:
"""Embed query text.

Args:
text: Text to embed.

Returns:
Embedding.
"""
return self.embed_documents([text])[0]

embedding = RemoteEmbedding(
base_url="https://dashscope.aliyuncs.com/compatible-mode/v1/embeddings", # 可
```

```
参考 https://bailian.console.aliyun.com/#/model-market/detail/text-embedding-v3?tabKey=sdk
api_key="your-api-key", # 填写你的 API Key
model="text-embedding-v3",
)

print("Embedding result:", embedding.embed_query("今天天气不错"), "\n")
# Embedding result: [-0.03573227673768997, 0.0645645260810852, ...]
print("Embedding results:", embedding.embed_documents(["今天天气不错", "明天呢?"]), "\n")
# Embedding results: [[-0.03573227673768997, 0.0645645260810852, ...],
[-0.05443647876381874, 0.07368793338537216, ...]]
```

5.2.3.5 使用通义千问 SDK

通义千问提供了名为 `dashscope` 的 SDK 用以快速调用模型能力，通过 `pip install dashscope` 安装完成后可获取到文本嵌入的内容。

```
import dashscope
from dashscope import TextEmbedding

# 设置 API Key
dashscope.api_key = "your-api-key"

# 准备输入文本
texts = ["这是第一句话", "这是第二句话"]

# 调用嵌入服务
response = TextEmbedding.call(
    model="text-embedding-v3",
    input=texts
```

```
)  
  
# 获取嵌入结果  
if response.status_code == 200:  
    print(response.output['embeddings'])  
# [{"embedding": [-0.03193652629852295, 0.08152323216199875, ...]},  
{"embedding": [...]}]
```


6 向量数据库迁移指南

OceanBase 提供了完整的向量数据迁移方案，帮助您轻松地将现有向量数据库中的数据迁移至 OceanBase 社区版。目前支持以下主流向量数据库的数据迁移：

6.1 Milvus 数据迁移

支持将 Milvus 数据库中的向量集合(Collection)、向量数据及其元数据无缝迁移至 OceanBase 社区版。迁移过程保证数据的完整性和一致性，让您的业务平滑过渡。请参见[迁移 Milvus 数据库的数据至 OceanBase 社区版](#)。

6.2 Qdrant 数据迁移

支持将 Qdrant 数据库中的向量集合(Collection)、向量点位(Points)及其 Payload 数据迁移至 OceanBase 社区版。迁移工具会自动处理数据格式转换，确保迁移后的数据可以直接使用。请参见[迁移 Qdrant 数据库的数据至 OceanBase 社区版](#)。

通过 OceanBase 的向量数据迁移能力,您可以：

- 统一数据管理,减少多个数据库带来的运维成本。
- 充分利用 OceanBase 的高性能向量检索能力。
- 实现数据库功能的无缝切换和升级。

7 向量检索实践教程概述

本文档将介绍如何基于 OceanBase 构建向量检索的典型应用场景。

7.1 向量检索实践教程

7.1.1 基于 OceanBase 构建智能问答机器人

文档智能助手是一种基于 OceanBase 数据库的应用，它将文档以向量的形式批量存储。用户可以通过 UI 界面向它提问，它会使用 BGE-M3 模型将提问内容嵌入成为向量，并在数据库中检索相似向量。当找到相似向量对应的文档内容后，应用会将它们会同用户提问一起发送给 LLM，LLM 会根据提供的文档生成更加准确的回答。

7.1.2 基于 OceanBase 构建图搜图应用

图搜图应用是另一种基于 OceanBase 数据库的应用，它将图片库以向量形式存储在数据库内。用户可以在对应的 UI 界面上传需要查询的图片，图片会被应用转换为向量，并在数据库内查询相似向量。最终，它会以图片形式，在 UI 页面上展示相似图片。

8 AI 集成概述

本文档将介绍如何基于 OceanBase 构建 AI 集成的典型应用场景。OceanBase 向量检索功能与各类 AI 框架和服务的集成，可以帮助你快速构建智能化应用。

8.1 AI 集成应用场景

在 AI 应用开发中，向量数据库是构建智能应用的重要基础设施。OceanBase 向量检索功能可以与主流的 AI 服务和框架无缝集成，支持以下典型应用场景：

8.1.1 OpenAI API 集成

OpenAI 提供了强大的语言模型和嵌入模型服务。通过集成 OpenAI API，你可以：

- 使用 OpenAI 的嵌入模型生成文本的向量表示。
- 将向量存储在 OceanBase 中进行高效检索。
- 结合 GPT 模型构建智能问答系统。
- 实现语义搜索等高级功能。

主要应用场景包括：

- 智能客服系统
- 文档检索系统
- 个性化推荐系统

8.1.2 通义千问 API 集成

通义千问是阿里云推出的大语言模型服务，支持中文场景的自然语言处理。通过集成通义千问 API，你可以：

- 使用通义千问的文本嵌入能力。
- 支持中文语义理解和检索。
- 构建垂直领域的智能应用。

适用场景：

- 企业知识库搜索
- 智能业务助手
- 多语言文档管理

8.1.3 Langchain 集成

Langchain 是一个用于开发 LLM 应用的框架。OceanBase 可以作为 Langchain 的向量存储后端，支持：

- 文档加载和向量化
- 构建对话检索链
- 实现 Agent 系统
- 知识库问答应用

集成优势：

- 简化 LLM 应用开发流程。
- 提供丰富的组件和工具。
- 支持灵活的应用定制。

8.1.4 LlamaIndex 集成

LlamaIndex 是专注于 LLM 应用中数据管理的框架。通过与 OceanBase 集成，可以：

- 高效管理和索引结构化数据。
- 支持复杂的数据查询和检索。
- 构建数据密集型的 AI 应用。

主要特点：

- 支持多种数据源接入。
- 提供数据更新和同步机制。
- 优化查询性能。

9 向量数据类型概述

OceanBase 提供向量数据类型以支撑 AI 向量检索的相关应用。通过使用向量数据类型，你可以存储和查询一个浮点数组，例如 `[0.1, 0.3, -0.9, ...]`。使用向量数据前，您需要知道以下事项：

- 向量数据类型支持的数据元素为单精度浮点数。
- 向量数据中的元素值不允许为 `NaN` 和 `Inf`，否则抛出运行时错误。
- 在创建向量列时必须指定向量维度，例如 `VECTOR(3)`。
- 支持创建向量索引，具体请参见[向量索引](#)。
- OceanBase 中的向量数据以数组的形式存储。

9.1 语法

一个向量值包含任意数量的浮点数。语法如下：

```
'[<float>, <float>, ...]'
```

创建向量列和索引示例如下：

```
CREATE TABLE t1(  
  c1 INT,  
  c2 VECTOR(3),  
  PRIMARY KEY(c1),  
  VECTOR INDEX idx1(c2) WITH (distance=L2, type=hns)  
);
```

10 向量函数

本文介绍了 OceanBase 支持的向量函数及使用注意事项。

10.1 使用注意事项

- 维度不同的向量数据不允许做下述运算，会报错 `different vector dimensions %d and %d`。
- 当结果超出浮点数值域时，会报错 `value out of range: overflow / underflow`。
- 支持 L2、内积（IP）、余弦距离作为索引距离算法。具体请参见[向量索引](#)。

10.2 距离函数

距离函数用于计算两个向量之间的距离，具体计算方式依据不同的距离算法而不同。

10.2.1 L2_distance

欧几里得距离反映的是被比较的向量坐标之间的距离--基本上是两个向量之间的直线距离。使用勾股定理应用于向量坐标来计算：

$$[D = \sqrt{\sum_{i=1}^n (x_{2,i} - x_{1,i})^2}]$$

函数语法如下：

```
l2_distance(vector v1, vector v2)
```

参数说明如下：

- 除了向量类型外，参数可接受任意可强转为向量类型的其他类型，包括单级数组类型（如 `[1,2,3,...]`）和字符串类型（如 `'[1,2,3]'`）。
- 两个参数的维度必须相同。
- 当存在单级数组类型参数时，该参数元素不允许出现 `NULL`。

返回值说明如下：

- 返回值为 `distance(double)` 距离值。
- 当任意参数为 `NULL` 时，返回 `NULL`。

示例如下：

```
CREATE TABLE t1(c1 vector(3));  
SELECT l2_distance(c1, [1,2,3]), l2_distance([1,2,3],[1,1,1]), l2_distance  
(['[1,1,1]','[1,2,3]') FROM t1;
```

10.2.2 L1_distance

曼哈顿距离用于计算两个点在标准的坐标系中的绝对轴距总和。计算公式为：
$$D = \sum_{i=1}^n |x_{1,i} - x_{2,i}|$$
。

函数语法如下：

```
l1_distance(vector v1, vector v2)
```

参数说明如下：

- 除了向量类型外，参数可接受任意可强转为向量类型的其他类型，包括单级数组类型（如 `[1,2,3,...]`）和字符串类型（如 `'[1,2,3]'`）。
- 两个参数的维度必须相同。
- 当存在单级数组类型参数时，该参数元素不允许出现 `NULL`。

返回值说明如下：

- 返回值为 `distance(double)` 距离值。
- 当任意参数为 `NULL` 时，返回 `NULL`。

示例如下：

```
CREATE TABLE t2(c1 vector(3));  
SELECT l1_distance(c1, [1,2,3]) FROM t2;
```

10.2.3 Cosine_distance

余弦相似度（cosine similarity）是衡量两个向量的角度差异，它反映了两个向量在方向上的相似度，与向量的长度（大小）无关。余弦相似度的取值范围为 `[-1, 1]`，其中 `1` 表示向量完全相同的方向，`0` 表示正交，`-1` 表示完全相反的方向。

余弦相似度的计算方式为：

$$[\text{Cosine Similarity} = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|} = \frac{\sum_{i=1}^n A_i B_i}{\sqrt{\sum_{i=1}^n A_i^2} \sqrt{\sum_{i=1}^n B_i^2}}]$$

由于余弦相似度度量越接近于 1 表示越相似，因此有时也使用余弦距离（或余弦不相似度）作为向量间距离的一种衡量方式，余弦距离可以通过 1 减去余弦相似度来计算：

$$[\text{Cosine Distance} = 1 - \text{Cosine Similarity}]$$

余弦距离的取值范围是 `[0, 2]`，其中 `0` 表示完全相同的方向（无距离），而 `2` 表示完全相反的方向。

函数语法如下：

```
cosine_distance(vector v1, vector v2)
```

参数说明如下：

- 除了向量类型外，参数可接受任意可强转为向量类型的其他类型，包括单级数组类型（如 `[1,2,3,...]`）和字符串类型（如 `'[1,2,3]'`）。
- 两个参数的维度必须相同。
- 当存在单级数组类型参数时，该参数元素不允许出现 `NULL`。

返回值说明如下：

- 返回值为 `distance(double)` 距离值。
- 当任意参数为 `NULL` 时，返回 `NULL`。

示例如下：

```
CREATE TABLE t3(c1 vector(3));
SELECT cosine_distance(c1, [1,2,3]) FROM t3;
```

10.2.4 Inner_product

内积又称为点积或数量积，表示两个向量之间的一种乘积。在几何意义上，内积表示两个向量的方向关系和大小关系。内积的计算方式为：

$$[\mathbf{A} \cdot \mathbf{B} = a_1 b_1 + a_2 b_2 + \dots + a_n b_n = \sum_{i=1}^n a_i b_i]$$

语法如下：

```
inner_product(vector v1, vector v2)
```

参数说明如下：

- 除了向量类型外，参数可接受任意可强转为向量类型的其他类型，包括单级数组类型（如 `[1,2,3,...]`）和字符串类型（如 `'[1,2,3]'`）。
- 两个参数的维度必须相同。
- 当存在单级数组类型参数时，该参数元素不允许出现 `NULL`。

返回值说明如下：

- 返回值为 `distance(double)` 距离值。
- 当任意参数为 `NULL` 时，返回 `NULL`。

示例如下：

```
CREATE TABLE t4(c1 vector(3));  
SELECT inner_product(c1, [1,2,3]) FROM t4;
```

10.2.5 Vector_distance

`vector_distance` 用于计算两个向量之间的距离，通过指定参数来选择不同的距离算法。

语法如下：

```
vector_distance(vector v1, vector v2 [, string metric])
```

`vector v1/v2` 参数说明如下：

- 除了向量类型外，参数可接受任意可强转为向量类型的其他类型，包括单级数组类型（如 `[1,2,3,...]`）和字符串类型（如 `'[1,2,3]'`）。
- 两个参数的维度必须相同。
- 当存在单级数组类型参数时，该参数元素不允许出现 `NULL`。

`metric` 参数用于指定距离算法，可选：

- 如果不指定，默认算法为 `euclidean`。
- 如果指定，可选择的值有且只有

- `euclidean` 。表示欧式距离，和 `L2_distance` 含义相同。
- `manhattan` 。表示曼哈顿距离，和 `L1_distance` 含义相同。
- `cosine` 。表示余弦距离，和 `Cosine_distance` 含义相同。
- `dot` 。表示内积，和 `Inner_product` 含义相同。

返回值说明如下：

- 返回值为 `distance(double)` 距离值。
- 当任意参数为 `NULL` 时，返回 `NULL` 。

示例如下：

```
CREATE TABLE t5(c1 vector(3));  
SELECT vector_distance(c1, [1,2,3], euclidean) FROM t5;
```

10.3 算术函数

算术函数提供向量类型与向量类型、单级数组类型、特殊字符串类型，以及单级数组类型与单级数组类型、特殊字符串类型的加（+）、减（-）和乘（*）法算术计算。计算方式为逐元素计算，如加法计算为：

$$[x_1, x_2, \dots, x_n] + [y_1, y_2, \dots, y_n] = [x_1 + y_1, x_2 + y_2, \dots, x_n + y_n]$$

语法如下：

`v1 + v2`

`v1 - v2`

`v1 * v2`

参数说明如下：

- 除了向量类型外，参数可接受任意可强转为向量类型的其他类型，包括单级数组类型（如 `[1,2,3,...]`）和字符串类型（如 `'[1,2,3]'`）。**注意：**两个参数不可以同时为字符串类型，必须有一个参数为向量或单级数组类型。
- 两个参数的维度必须相同。
- 当存在单级数组类型参数时，该参数元素不允许出现 `NULL`。

返回值说明如下：

- 当两个参数中至少有一个参数为向量类型时，返回值为和向量参数相同的向量类型。
- 当两个参数中只有单级数组类型时，返回值为 `array(float)` 类型。
- 当任意参数为 `NULL` 时，返回 `NULL`。

示例如下：

```
CREATE TABLE t6(c1 vector(3));  
SELECT [1,2,3] + '[1.12,1000.0001, -1.2222]', c1 - [1,2,3] FROM t6;
```

10.4 比较函数

比较函数提供向量类型与向量类型、单级数组类型、特殊字符串类型的比较计算，包括 `=`、`!=`、`>`、`<`、`>=`、`<=` 几种比较符号。计算方式为逐元素的字典序比较。

语法如下：

```
v1 = v2  
v1 != v2  
v1 > v2  
v1 < v2  
v1 >= v2  
v1 <= v2
```

参数说明如下：

- 除了向量类型外，参数可接受任意可强转为向量类型的其他类型，包括单级数组类型（如 `[1,2,3,...]`）和字符串类型（如 `'[1,2,3]'`）。

10.4.5.1 注意

两个参数必须有一个参数为向量类型。

- 两个参数的维度必须相同。
- 当存在单级数组类型参数时，该参数元素不允许出现 `NULL`。

返回值说明如下：

- 返回值为 `bool` 类型。
- 当任意参数为 `NULL` 时，返回 `NULL`。

示例如下：

```
CREATE TABLE t7(c1 vector(3));  
SELECT c1 = '[1,2,3]' FROM t7;
```

10.5 聚合函数

10.5.5.1 注意

不支持使用向量列作为 GROUP BY 条件，不支持 DISTINCT。

10.5.6 Sum

Sum 函数用于计算表中向量列的和，采用逐元素累加的方式进行计算，以得到和向量。

语法如下：

```
sum(vector v1)
```

参数说明如下：

- 仅支持向量类型。

返回值说明如下：

- 返回 `sum (vector)` 值。

示例如下：

```
CREATE TABLE t8(c1 vector(3));  
SELECT sum(c1) FROM t8;
```

10.5.7 Avg

Avg 函数用于计算表中向量列的平均值。

语法如下：

```
avg(vector v1)
```

参数说明如下：

- 仅支持向量类型。

返回值说明如下：

- 返回 `avg (vector)` 值。

- 不计入该向量列中的 `NULL` 行。
- 输入参数为空时，输出 `NULL`。

示例如下：

```
CREATE TABLE t9(c1 vector(3));  
SELECT avg(c1) FROM t9;
```

10.6 其他常见向量函数

10.6.8 Vector_norm

`Vector_norm` 函数用于计算向量的欧几里得范数（模），表示向量和原点之间的欧几里得距离。计算方式为：

$$[D = \sqrt{\sum_{i=1}^n (x_i)^2}]$$

语法如下：

```
vector_norm(vector v1)
```

参数说明如下：

- 除了向量类型外，参数可接受任意可强转为向量类型的其他类型，包括单级数组类型（如 `[1,2,3,..]`）和字符串类型（如 `'[1,2,3]'`）。
- 当存在单级数组类型参数时，该参数元素不允许出现 `NULL`。

返回值说明如下：

- 返回 `norm(double)` 模值。
- 当参数为 `NULL` 时，返回 `NULL`。

示例如下：

```
CREATE TABLE t10(c1 vector(3));  
SELECT vector_norm(c1),vector_norm([1,2,3]) FROM t10;
```

10.6.9 Vector_dims

`Vector_dims` 函数用于返回向量维度。

语法如下：

```
vector_dims(vector v1)
```

参数说明如下：

- 除了向量类型外，参数可接受任意可强转为向量类型的其他类型，包括单级数组类型（如 [1,2,3,..]）和字符串来类型（如 '[1,2,3]'）。

返回值说明如下：

- 返回 dims(int64) 维度值。
- 当参数为 NULL 时，返回 NULL。

示例如下：

```
CREATE TABLE t11(c1 vector(3));  
SELECT vector_dims(c1),vector_dims('[1,2,3]'),vector_dims([1,null,3]) FROM t11;
```

11 向量索引

本文介绍了 OceanBase 如何创建、查询、维护和删除向量索引。

OceanBase 向量索引的具体说明如下：

- 支持 HNSW 索引，索引列最大维度为 4096。HNSW 索引是一种内存索引，需要完整载入内存，支持 DML 和实时查询。
- 支持 L2、内积（IP）、余弦距离作为索引距离算法。

11.0.0.1 注意

使用向量索引前，需要通过设置 [ob_vector_memory_limit_percentage](#) 开启向量功能。

11.1 索引查询概述

向量索引查询是一种近似最近邻查询，并不保证 100% 的结果正确。相应的向量查询准确率的指标是召回率，例如在查 10 个最近邻时，如果可以稳定返回 9 个正确的结果，那么召回率就是 90%。召回率说明如下：

- 召回率受构建参数和查询参数的影响。
- 查询参数在建索引时指定，之后不可修改，但可通过 session 变量 `ob_hnsw_ef_search` 设置。如果设置了 `ob_hnsw_ef_search`，会优先使用它的值。具体设置方式请参见 [ob_hnsw_ef_search](#)。

查询语法如下：

```
SELECT ... FROM $table_name ORDER BY $distance_function($column_name,
$vector_expr) [APPROXIMATE|APPROX] LIMIT|OFFSET $num;
```

使用说明如下：

- 目前仅支持 `ORDER BY` 后跟随一个向量条件的情况，否则会报错 `ERROR 1235 (0A000): Multi order by item when using vector index is not supported`。
- 查询包含 `WHERE` 条件时，不论条件是否为其他索引列，只要满足上述语法都会调用向量索引，`WHERE` 条件作为向量索引查询后的过滤条件。
- 必须带有 `APPROXIMATE / APPROX` 关键字，查询计划才会选择向量索引路径。

- 支持 `l2_distance` , `inner_product` , `negative_inner_product` 作为 `ORDER BY` 指定距离函数, 其中 `l2_distance` , `negative_inner_product` 选择索引计划, `inner_product` 不选择索引计划。
- 除了 `l2_distance` , `inner_product` , `negative_inner_product` 以外的距离函数, 使用 `APPROXIMATE` / `APPROX` 选择向量索引路径时会报错 `ERROR 1235 (0A000): Using vector index without vector_sort_expr is not supported` 。
- `LIMIT` / `OFFSET` 的取值范围为 `(0, 16384]` 。

11.2 索引创建、查询及删除示例

11.2.1 注意事项

OceanBase 向量索引的创建支持在建表时创建和后建两种方式。创建索引时需要注意:

- 向量索引创建后, 不支持任何参数的修改。
- 创建向量索引必须带有 `VECTOR` 关键字。
- 后建索引的参数和说明与建表时创建索引一致。
- 如果数据量较大, 建议先写完数据, 再创建索引, 以获得最佳查询性能。

11.2.2 建表时创建索引

创建测试表。

```
CREATE TABLE t1(c1 INT, c0 INT, c2 VECTOR(10), c3 VECTOR(10), PRIMARY KEY(c1),  
VECTOR INDEX idx1(c2) WITH (distance=l2, type=hns, lib=vsag), VECTOR INDEX idx2  
(c3) WITH (distance=l2, type=hns, lib=vsag));
```

写入测试数据。

```
INSERT INTO t1 VALUES(1, 1, '[0.203846,0.205289,0.880265,0.824340,0.615737,0.4968  
99,0.983632,0.865571,0.248373,0.542833]', '[0.203846,0.205289,0.880265,0.824340,0  
.615737,0.496899,0.983632,0.865571,0.248373,0.542833]');
```

```
INSERT INTO t1 VALUES(2, 2, '[0.735541,0.670776,0.903237,0.447223,0.232028,0.6593
```



```
16,0.765661,0.226980,0.579658,0.933939]', '[0.213846,0.205289,0.880265,0.824340,0.615737,0.496899,0.983632,0.865571,0.248373,0.542833]');
```

```
INSERT INTO t1 VALUES(3, 3, '[0.327936,0.048756,0.084670,0.389642,0.970982,0.370915,0.181664,0.940780,0.013905,0.628127]', '[0.223846,0.205289,0.880265,0.824340,0.615737,0.496899,0.983632,0.865571,0.248373,0.542833]');
```

使用近似最近邻查询。

```
SELECT * FROM t1 ORDER BY l2_distance(c2, [0.712338,0.603321,0.133444,0.428146,0.876387,0.763293,0.408760,0.765300,0.560072,0.900498]) APPROXIMATE LIMIT 1;
```

返回结果如下：

```
+----+-----+-----+-----+
-----+-----+
-----+
| c1 | c0 | c2 | c3 |
+----+-----+-----+-----+
-----+-----+
-----+
| 3 | 3 | [0.327936,0.048756,0.08467,0.389642,0.970982,0.370915,0.181664,0.94078,0.013905,0.628127] | [0.223846,0.205289,0.880265,0.82434,0.615737,0.496899,0.983632,0.865571,0.248373,0.542833] |
+----+-----+-----+-----+
-----+-----+
-----+
1 row in set
```

索引参数说明：

参数	默认值	取值范围	是否必填	说明	备注
distance		l2/inner_product	是	指定向量距离函数类型。	l2 表示欧氏距离，inner_product 表示内积距离。
type		hnsw	是	指定索引类型。	目前仅支持 HNSW 索引。
lib	vsag	vsag	否	指定向量索引库类型。	目前仅支持 VSAG 向量库。
m	16	[5,64]	否	HNSW 构建参数，每个节点的最大邻居数。	值越大，索引构建越慢，查询性能越好。
ef_construction	200	[5,1000]	否	HNSW 构建参数，构建索引时的候选集大小。	值越大，索引构建越慢，索引质量越好。ef_construction 必须大于 m。
ef_search	64	[1,1000]	否	HNSW 查询参数，查询时的候选集大小。	值越大，查询越慢，召回率越高。

11.2.3 后建索引

创建测试表。

```
CREATE TABLE vec_table (id int, c2 vector(10));
```

创建向量索引。

```
CREATE VECTOR INDEX vec_idx1 ON vec_table(c2) WITH (distance=l2, type=hnsw);
```

查看创建的表。

```
SHOW CREATE TABLE vec_table;
```

返回结果如下：

+	-----	+
+	-----	+

```
-----+
| Table | Create Table |
+-----+-----+
-----+
-----+
-----+
-----+
-----+
-----+
-----+
| vec_table | CREATE TABLE `vec_table` (
`id` int(11) DEFAULT NULL,
`c2` VECTOR(10) DEFAULT NULL,
VECTOR KEY `vec_idx1` (`c2`) WITH (DISTANCE=L2, TYPE=HNSW, LIB=VSAG, M=16,
EF_CONSTRUCTION=200, EF_SEARCH=64) BLOCK_SIZE 16384
) DEFAULT CHARSET = utf8mb4 ROW_FORMAT = DYNAMIC COMPRESSION = 'zstd_1.3.8'
REPLICA_NUM = 2 BLOCK_SIZE = 16384 USE_BLOOM_FILTER = FALSE TABLET_SIZE =
134217728 PCTFREE = 0 |
+-----+-----+
-----+
-----+
-----+
-----+
-----+
-----+
-----+
1 row in set

OceanBase(root@oceanbase)>SHOW INDEX FROM vec_table;
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
+-----+-----+
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation |
```

```

Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment | Visible
| Expression |
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+
| vec_table | 1 | vec_idx1 | 1 | c2 | A | NULL | NULL | NULL | YES | VECTOR | available | YES |
NULL |
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+
1 row in set

```

11.2.4 删除索引

删除向量索引。

```
DROP INDEX vec_idx1 ON vec_table;
```

查看删除的索引。

```
SHOW INDEX FROM vec_table;
```

返回结果如下：

```
Empty set
```

11.3 索引维护

增量数据过多的情况下，查询性能会下降。为减小增量数据表的数据量，OceanBase 引入了 `DBMS_VECTOR` 对向量索引进行维护。

11.3.5 增量刷新

如果建立索引后写入数据较多，建议使用增量刷新。

语法如下：

```
PROCEDURE refresh_index(  
IN idx_name VARCHAR(65535), ---- 索引名  
IN table_name VARCHAR(65535), ---- 表名  
IN idx_vector_col VARCHAR(65535) DEFAULT NULL, ---- 向量列名  
IN refresh_threshold INT DEFAULT 10000, ---- 增量数据数量  
IN refresh_type VARCHAR(65535) DEFAULT NULL); ---- FAST
```

增量刷新每 15 分钟会检查一次，如果增量数据超过 1W 条，则自动执行增量刷新。

11.3.6 全量刷新

如果建立索引后更新或删除数据较多，建议使用全量刷新。

语法如下：

```
PROCEDURE rebuild_index(  
IN idx_name VARCHAR(65535), ---- 索引名  
IN table_name VARCHAR(65535), ---- 表名  
IN idx_vector_col VARCHAR(65535) DEFAULT NULL, ---- 向量列名  
IN delta_rate_threshold FLOAT DEFAULT 0.2, ---- 增量数据的比例  
IN idx_organization VARCHAR(65535) DEFAULT NULL, ---- 索引类型  
IN idx_distance_metrics VARCHAR(65535) DEFAULT 'EUCLIDEAN', ---- 距离类型  
IN idx_parameters LONGTEXT DEFAULT NULL, ---- 索引参数  
IN idx_parallel_creation INT DEFAULT 1); ---- 并行构建索引的并行度，预留，仅语法支持
```

全量刷新每 24 小时会检查一次，如果新增数据超过原有数据的 20%，则自动执行全量刷新。

11.3.6.1 索引维护示例

创建测试表。

```
CREATE TABLE vector_index_test(c1 INT, c2 VECTOR(3), PRIMARY KEY(c1), VECTOR  
INDEX idx1(c2) WITH (distance=l2, type=hns, lib=vsag));
```

写入测试数据。

```
INSERT INTO vector_index_test VALUES(1, '[0.203846,0.205289,0.880265]');
INSERT INTO vector_index_test VALUES(2, '[0.484526,0.669954,0.986755]');
INSERT INTO vector_index_test VALUES(3, '[0.327936,0.048756,0.084670]');
INSERT INTO vector_index_test VALUES(4, '[0.148869,0.878546,0.028024]');
INSERT INTO vector_index_test VALUES(5, '[0.334970,0.857377,0.886132]');
INSERT INTO vector_index_test VALUES(6, '[0.117582,0.302352,0.471198]');
INSERT INTO vector_index_test VALUES(7, '[0.551185,0.231134,0.075354]');
INSERT INTO vector_index_test VALUES(8, '[0.185221,0.315131,0.558301]');
INSERT INTO vector_index_test VALUES(9, '[0.928764,0.254038,0.272721]');
```

触发更新。

```
-- 触发增量更新。
```

```
call dbms_vector.refresh_index('idx1', 'vector_index_test', 'c2', 1, 'FAST');
```

```
-- 触发全量更新。
```

```
call dbms_vector.rebuild_index('idx1','t1','c2');
```

11.4 相关文档

- [向量函数](#)

12 兼容性说明

本文档介绍了 OceanBase 向量检索功能与 Milvus 的数据模型映射关系、SDK 接口兼容性以及相关概念对照说明。

12.1 与 Milvus 概念映射

为方便熟知 Milvus 的用户上手 OceanBase 向量存储能力，分析两者之间的异同并提供相关概念之间的映射。

12.1.1 数据模型

数据模型层级	Milvus	OceanBase	描述
第一层	Shards	Partition	Milvus 通过在模式定义时设置某些列为 <code>partition_key</code> 来制定分区规则 OceanBase 支持 <code>range /range columns</code> 、 <code>list /list columns</code> 、 <code>hash</code> 、 <code>key</code> 以及二级分区策略
第二层	Partitions	≈Tablet	Milvus 支持在同一个 Shards 中（Shards 通常按照主键进行分区）按照其他列进行分块，以提升读性能 OceanBase 则是通过在分区内按照主键排序实现
第三层	Segments	MemTable+SSTable	两者都有转储行为

12.1.2 SDK

本小节介绍 OceanBase 向量存储的 SDK —— `pyobvector` 和 Milvus 的 SDK —— `pymilvus` 的概念差异。

`pyobvector` 具有两种使用模式：

1. `pymilvus MilvusClient` 轻量兼容模式：该模式兼容了 Milvus 客户端常用的接口，熟悉 Milvus 的用户非常方便地使用本模式，无需概念映射。
2. `SQLAlchemy` 扩展模式：该模式可以作为 python `SQLAlchemy` 的向量功能扩展，保留了关系型数据库的操作模式，需要概念映射进行说明。

pyobvector 具体接口使用文档请参见 [pyobvector Python SDK 接口说明](#)。

下表说明 pyobvector SQLAlchemy 扩展模式下和 Milvus SDK 接口的概念映射：

pymilvus	pyobvector	描述
Database	Database	数据库
Collection	Table	表
Field	Column	列
Primary Key	Primary Key	主键
Vector Field	Vector Column	向量列
Index	Index	索引
Partition	Partition	分区
DataType	DataType	数据类型
Metric Type	Distance Function	距离函数
Search	Query	查询
Insert	Insert	插入
Delete	Delete	删除
Update	Update	更新
Batch	Batch	批量操作
Transaction	Transaction	事务
NONE	不支持	NULL 值
BOOL	Boolean	对应 MySQL TINYINT 类型
INT8	Boolean	对应 MySQL TINYINT 类型
INT16	SmallInteger	对应 MySQL SMALLINT 类型
INT32	Integer	对应 MySQL INT 类型
INT64	BigInteger	对应 MySQL BIGINT 类型
FLOAT	Float	对应 MySQL FLOAT 类型
DOUBLE	Double	对应 MySQL DOUBLE 类型
STRING	LONGTEXT	对应 MySQL LONGTEXT 类型
VARCHAR	STRING	对应 MySQL VARCHAR 类型
JSON	JSON	JSON 操作异同点请参见 pyobvector Python SDK 接口说明

FLOAT_VECTOR	VECTOR	向量类型
BINARY_VECTOR	不支持	
FLOAT16_VECTOR	不支持	
BFLOAT16_VECTOR	不支持	
SPARSE_FLOAT_VECTOR	不支持	
dynamic_field	无需支持	Milvus 中的隐藏 <code>\$meta</code> 元数据列 OceanBase 显示创建 JSON 类型的列即可

12.2 与 Milvus 兼容性

12.2.3 Milvus SDK

除 `load_collection()` / `release_collection()` / `close()` 等需通过 SQLAlchemy 支持外，下述表中所列接口均可兼容：

Collection 操作

接口	描述
<code>create_collection()</code>	根据给定的 Schema 创建一个向量表。
<code>get_collection_stats()</code>	查看数据表统计信息，如记录行数。
<code>describe_collection()</code>	提供向量表的详细元信息。
<code>has_collection()</code>	判断表是否存在。
<code>list_collections()</code>	列出存在的表。
<code>drop_collection()</code>	删除表。

Field 与 Schema 定义

接口	描述
<code>create_schema()</code>	创建一个 Schema 内存结构、添加列定义。
<code>add_field()</code>	使用时调用序列 <code>create_schema->add_field->...->add_field</code> 也可以手动构造一个 <code>FieldSchema</code> 列表，再通过 <code>CollectionSchema</code> 的构造函数来创建 Schema。

向量索引

接口	描述
list_indexes()	列出所有索引。
create_index()	支持一次调用创建多个向量索引，首先通过 prepare_index_params 来初始化一个索引参数列表对象，调用多次 add_index 来设置多个索引参数，最终通过 create_index 来创建索引。
drop_index()	删除向量索引。
describe_index()	获得索引的元数据（Schema）。

向量索引

接口	描述
search()	ANN 查询接口： <ul style="list-style-type: none">• collection_name: 表名• data: 查询的向量• filter: 筛选操作，等价于 WHERE• limit: top K• output_fields: 投影列，等价于 SELECT• partition_names: 分区名（Milvus Lite 不支持）• anns_field: 索引列名• search_params: 向量距离函数名、索引算法相关参数
query()	含 filter 点查询，即 SELECT ... WHERE ids IN (... , ...) AND <filters>。
get()	不含 filter 点查询，即 SELECT ... WHERE ids IN (... , ...)。
delete()	删除一组向量，DELETE FROM ... WHERE ids IN (... , ...)。
insert()	插入一组向量。
upsert()	主键冲突带更新的插入。

collection 元数据同步

接口	描述
<code>load_collection()</code>	将数据库中的表结构(Schema)加载到 Python 应用程序内存中，使应用程序能够以面向对象的方式操作数据库表。是对象关系映射（ORM）框架的一个标准功能。
<code>release_collection()</code>	从 Python 应用程序内存中释放已加载的表结构(Schema)，释放相关资源。这是 ORM 框架的一个标准功能,用于内存管理。
<code>close()</code>	关闭数据库连接，释放相关资源。这是 ORM 框架的标准功能。

12.2.4 pymilvus

12.2.4.1 数据模型

Milvus 的数据模型分为 Shards->Partitions->Segments 三个级别，与 OceanBase 兼容性对比如下：

- Shards 对应于 OceanBase 的 Partition 概念。
- Partitions 目前 OceanBase 无对应概念。
- Milvus 支持在同一个 Shards 中（Shards 通常按照主键进行分区）按照其他列进行分块，以提升读性能。OceanBase 则是通过在分区内按照主键排序实现。
- Segments 类似于 [MemTable](#) + [SSTable](#)。

12.2.4.2 Milvus Lite API 兼容性

collection 操作

1. Milvus `create_collection()`:

```
create_collection(  
    collection_name: str,  
    dimension: int,  
    primary_field_name: str = "id",  
    id_type: str = DataType,  
    vector_field_name: str = "vector",
```

```
metric_type: str = "COSINE",
auto_id: bool = False,
timeout: Optional[float] = None,
schema: Optional[CollectionSchema] = None, # Used for custom setup
index_params: Optional[IndexParams] = None, # Used for custom setup
**kwargs,
) -> None
```

OceanBase 兼容说明如下：

- collection_name: 兼容，实际为 table_name。
- dimension: 兼容，vector(dim)。
- primary_field_name: 兼容，主键列名。
- id_type: 兼容，主键列列类型。
- vector_field_name: 兼容，向量列名。
- auto_id: 兼容，auto increment。
- timeout: 兼容，OceanBase通过hint支持。
- schema: 兼容。
- index_params: 兼容。

2. Milvus get_collection_stats():

```
get_collection_stats(
collection_name: str,
timeout: Optional[float] = None
) -> Dict
```

OceanBase 兼容说明如下：

- API 兼容。
- 返回值兼容： { 'row_count': ... } 。

3. Milvus has_collection():

```
has_collection(
collection_name: str,
```

```
timeout: Optional[float] = None
) -> Bool
```

OceanBase 兼容 Milvus has_collection()。

4. Milvus drop_collection():

```
drop_collection(collection_name: str) -> None
```

OceanBase 兼容 Milvus drop_collection()。

5. Milvus rename_collection():

```
rename_collection(
old_name: str,
new_name: str,
timeout: Optional[float] = None
) -> None
```

OceanBase 兼容 Milvus rename_collection()。

Schema 相关

1. Milvus create_schema():

```
create_schema(
auto_id: bool,
enable_dynamic_field: bool,
primary_field: str,
partition_key_field: str,
) -> CollectionSchema
```

OceanBase 兼容说明如下：

- auto_id：是否自增主键列，可兼容。
- primary_field & partition_key_field：可兼容。

2. Milvus add_field():

```
add_field(  
    field_name: str,  
    datatype: DataType,  
    is_primary: bool,  
    max_length: int,  
    element_type: str,  
    max_capacity: int,  
    dim: int,  
    is_partition_key: bool,  
)
```

OceanBase 兼容 Milvus add_field()。

Insert/Search 相关

1. Milvus search():

```
search(  
    collection_name: str,  
    data: Union[List[list], list],  
    filter: str = "",  
    limit: int = 10,  
    output_fields: Optional[List[str]] = None,  
    search_params: Optional[dict] = None,  
    timeout: Optional[float] = None,  
    partition_names: Optional[List[str]] = None,  
    **kwargs,  
) -> List[dict]
```

OceanBase 兼容说明如下：

- filter: str 表达式，具体使用示例参见：<https://milvus.io/docs/boolean.md#Usage>，大体上和 SQL 的 WHERE 表达式类似。
- search_params:

- `metric_type`: 兼容。
- `radius & range filter`: RNN 相关, 目前不支持。
- `group_by_field`: 对 ANN 结果进行分组, 目前不支持。
- `max_empty_result_buckets`: IVF 系列索引使用, 目前不支持。
- `ignore_growing`: 跳过增量部分, 直接读取基线索引, 目前不支持。
- `partition_names`: 分区读, 支持。
- `kwargs`:
 - `offset`: 搜索结果中要跳过的记录数, 目前不支持。
 - `round_decimal`: 对结果进行指定小数位数的四舍五入, 目前不支持。

2. Milvus `get()`:

```
get(
collection_name: str,
ids: Union[list, str, int],
output_fields: Optional[List[str]] = None,
timeout: Optional[float] = None,
partition_names: Optional[List[str]] = None,
**kwargs,
) -> List[dict]
```

OceanBase 兼容 Milvus `get()`。

3. Milvus `delete()`

```
delete(
collection_name: str,
ids: Optional[Union[list, str, int]] = None,
timeout: Optional[float] = None,
filter: Optional[str] = "",
partition_name: Optional[str] = "",
**kwargs,
) -> dict
```

OceanBase 兼容 Milvus `delete()`。

4. Milvus insert()

```
insert(  
collection_name: str,  
data: Union[Dict, List[Dict]],  
timeout: Optional[float] = None,  
partition_name: Optional[str] = "",  
) -> List[Union[str, int]]
```

OceanBase 兼容 Milvus insert()。

5. Milvus upsert()

```
upsert(  
collection_name: str,  
data: Union[Dict, List[Dict]],  
timeout: Optional[float] = None,  
partition_name: Optional[str] = "",  
) -> List[Union[str, int]]
```

OceanBase 兼容 Milvus upsert()。

index 相关

1. Milvus create_index()

```
create_index(  
collection_name: str,  
index_params: IndexParams,  
timeout: Optional[float] = None,  
**kwargs,  
)
```

OceanBase 兼容 Milvus create_index()。

2. Milvus drop_index()


```
drop_index(  
collection_name: str,  
index_name: str,  
timeout: Optional[float] = None,  
**kwargs,  
)
```

OceanBase 兼容 Milvus drop_index()。

12.3 与 MySQL 协议兼容性

- request 请求发起方面：API 均通过普通查询 SQL 来实现，不存在兼容性问题。
- response 结果集处理方面：只需考虑新增的向量数据元素的处理，目前支持 string 和 bytes 两种元素的解析，即使后续向量数据元素传输方式发生变化，也可通过更新 SDK 完成兼容。

13 pyobvector Python SDK 接口说明

pyobvector 是 OceanBase 向量存储功能的 python SDK，它提供两种使用模式：

- pymilvus 兼容模式：使用 MilvusLikeClient 对象操作数据库，提供与轻量级 MilvusClient 兼容的常用接口。
- SQLAlchemy 扩展模式：使用 ObVecClient 对象操作数据库，提供关系型数据库的 python SDK 扩展。

本文分别介绍了这两种模式下的使用接口与示例。

13.1 MilvusLikeClient

13.1.1 构造函数

```
def __init__(
    self,
    uri: str = "127.0.0.1:2881",
    user: str = "root@test",
    password: str = "",
    db_name: str = "test",
    **kwargs,
)
```

13.1.2 collection 相关接口

API 接口	参数描述	示例
--------	------	----

<pre>def create_schema(self, **kwargs) -> CollectionSchema:</pre>	<p>构造一个 <code>CollectionSchema</code> 对象</p> <ul style="list-style-type: none">• 可以不传参数，即初始化一个空的模式定义。• 可选参数如下：<ul style="list-style-type: none">• <code>fields</code>: 一个 <code>FieldSchema</code> 列表（详见下文 <code>add_schema</code> 接口）• <code>partitions</code>: 分区规则（详见使用 <code>ObPartition</code> 定义分区规则章节）• <code>description</code>: 与 Milvus 兼容用，在 OceanBase 中暂无实际作用	
--	--	--

<pre>def create_collection(self, collection_name: str, dimension: Optional[int] = None, primary_field_name: str = "id", id_type: Union[DataType, str] = DataType.INT64, vector_field_name: str = "vector", metric_type: str = "l2", auto_id: bool = False, timeout: Optional[float] = None, schema: Optional [CollectionSchema] = None, # Used for custom setup index_params: Optional [IndexParams] = None, # Used for custom setup max_length: int = 16384, **kwargs,)</pre>	<p>创建一个表：</p> <ul style="list-style-type: none"> • collection_name : 表名称 • dimension : 向量数据维度 • primary_field_name: 主字段名称 • id_type: 主字段数据类型(仅支持 VARCHAR 和 INT 类型) • vector_field_name : 向量字段名称 • metric_type: OceanBase 中未使用，但保持接口兼容 (因为主表定义不需要指定向量距离函数) • auto_id: 主字段是否自动递增 • timeout : OceanBase 中未使用，但保持接口兼容 • schema : 自定义集合架构，当 schema 不为 None 时上面从 dimension 到 metric_type 的参数将被忽略 • index_params: 自定义向量索引参数 • max_length: 当主字段数据类型为 VARCHAR 且 schema 不为 None 时最大 varchar 长度为 max_length 	<pre>client.create_collection(collection_name=test_collectio n_name, schema=schema, index_params=idx_params,)</pre>
<pre>def get_collection_stats(self, collection_name: str, timeout: Optional[float] = None # pylint: disable=unused-argument) -> Dict:</pre>	<p>获取表的记录数量</p> <ul style="list-style-type: none"> • collection_name: 表名称 • timeout : OceanBase 中未使用，但保持接口兼容 	

<pre>def has_collection(self, collection_name: str, timeout: Optional[float] = None) -> bool</pre>	判断表是否存在 <ul style="list-style-type: none"> • collection_name: 表名称 • timeout : OceanBase 中未使用, 但保持接口兼容 	
<pre>def drop_collection(self, collection_name: str) -> None</pre>	重命名表 <ul style="list-style-type: none"> • old_name: 表的原名 • new_name: 新表名 	
<pre>def load_table(self, collection_name: str,)</pre>	读取表元数据到 SQLAlchemy 元数据缓存 <ul style="list-style-type: none"> • collection_name: 表名称 	

13.1.3 CollectionSchema & FieldSchema

MilvusLikeClient 通过 CollectionSchema 描述一个表的模式定义, 一个 CollectionSchema 包含多个 FieldSchema, FieldSchema 描述一个表的列模式。

13.1.3.1 通过 MilvusLikeClient 的 create_schema 创建 CollectionSchema

```
def __init__(
self,
fields: Optional[List[FieldSchema]] = None,
partitions: Optional[ObPartition] = None,
description: str = "", # ignored in oceanbase
**kwargs,
)
```

参数说明如下:

- fields: 一组可选的 FieldSchema。
- partitions: 分区规则（详见使用 ObPartition 定义分区规则章节）。
- description: 与 Milvus 兼容用, 在 OceanBase 中暂无实际作用。

13.1.3.2 创建 FieldSchema 并注册到 CollectionSchema

```
def add_field(self, field_name: str, datatype: DataType, **kwargs)
```

- field_name: 列名称。
- datatype: 列数据类型（支持的数据类型请参见[兼容性说明](#)）。
- kwargs: 其他参数用于配置列属性，如下：

```
def __init__(
    self,
    name: str,
    dtype: DataType,
    description: str = "",
    is_primary: bool = False,
    auto_id: bool = False,
    nullable: bool = False,
    **kwargs,
)
```

参数说明如下：

- is_primary: 是否是主键。
- auto_id: 是否是自增列。
- nullable: 是否允许为空。

13.1.3.3 使用示例

```
schema = self.client.create_schema()
schema.add_field(field_name="id", datatype=DataType.INT64, is_primary=True)
schema.add_field(field_name="title", datatype=DataType.VARCHAR,
max_length=512)
schema.add_field(
```

```
field_name="title_vector", datatype=DataType.FLOAT_VECTOR, dim=768
)
schema.add_field(field_name="link", datatype=DataType.VARCHAR,
max_length=512)
schema.add_field(field_name="reading_time", datatype=DataType.INT64)
schema.add_field(
field_name="publication", datatype=DataType.VARCHAR, max_length=512
)
schema.add_field(field_name="claps", datatype=DataType.INT64)
schema.add_field(field_name="responses", datatype=DataType.INT64)

self.client.create_collection(
collection_name="medium_articles_2020", schema=schema
)
```

13.1.4 索引相关

API 接口	参数描述	示例或备注
<pre>def create_index(self, collection_name: str, index_params: IndexParams, timeout: Optional[float] = None, **kwargs,)</pre>	<p>根据已构造的 IndexParams 创建向量索引表（此接口关于 IndexParams 的使用详见 prepare_index_params 和 add_index 接口）</p> <ul style="list-style-type: none">• collection_name: 表名称• index_params: 索引参数• timeout: OceanBase 中未使用，但保持接口兼容• kwargs: 其他参数，目前未使用，保持兼容	

<pre>def drop_index(self, collection_name: str, index_name: str, timeout: Optional[float] = None, **kwargs,)</pre>	<p>删除索引表</p> <ul style="list-style-type: none"> • collection_name: 表名称 • index_name: 索引名 	
<pre>def refresh_index(self, collection_name: str, index_name: str, trigger_threshold: int = 10000,)</pre>	<p>刷新向量索引表以提升读取性能，可以理解为对增量数据的搬迁</p> <ul style="list-style-type: none"> • collection_name: 表名称 • index_name: 索引名 • trigger_threshold: 刷新动作的触发阈值，如果索引表数据量超过该阈值，则进行刷新 	<p>OceanBase 额外引入的接口 非 Milvus 兼容</p>
<pre>def rebuild_index(self, collection_name: str, index_name: str, trigger_threshold: float = 0.2,)</pre>	<p>重建向量索引表以提升读取性能，可以理解为将增量数据合并入基线索引数据</p> <ul style="list-style-type: none"> • collection_name: 表名称 • index_name: 索引名 • trigger_threshold: 重建动作的触发阈值，值域 0 到 1，增量数据占全量数据比例达到该阈值时触发重建 	<p>OceanBase 额外引入的接口 非 Milvus 兼容</p>

<pre>def search(self, collection_name: str, data: list, anns_field: str, with_dist: bool = False, filter=None, limit: int = 10, output_fields: Optional[List [str]] = None, search_params: Optional[dict] = None, timeout: Optional[float] = None, partition_names: Optional[List [str]] = None, **kwargs,) -> List[dict]</pre>	<p>执行向量近似邻近搜索</p> <ul style="list-style-type: none"> • collection_name: 表名称 • data: 需要搜索的向量数据 • anns_field: 需要搜索的向量列名 • with_dist: 是否返回带向量距离的结果 • filter : 使用带过滤条件的向量近似邻近搜索 • limit : top K • output_fields: 输出列（或称为投影列） • search_params : 仅支持值为 l2/neg_ip 的 metric_type（例如：search_params = {"metric_type": "neg_ip"}） • timeout : OceanBase 中未使用，仅兼容作用 • partition_names : 将查询限制在某些分区 <p>返回值： 记录列表，每条记录都是一个字典 表示从 column_name 到列值的映射。</p>	<pre>res = self.client.search(collection_name=test_collectio n_name, data=[0, 0, 1], anns_field="embedding", limit=5, output_fields=["id"], search_params= {"metric_type": "neg_ip"}) self.assertEqual(set([r['id'] for r in res]), set([12, 111, 11, 112, 10]))</pre>
--	--	---

<pre>def query(self, collection_name: str, filter=None, output_fields: Optional[List [str]] = None, timeout: Optional[float] = None, partition_names: Optional[List [str]] = None, **kwargs,) -> List[dict]</pre>	<p>使用指定过滤条件读取数据记录</p> <ul style="list-style-type: none">• collection_name: 表名称• filter : 使用带过滤条件的向量近似邻近搜索• output_fields: 输出列（或称为投影列）• timeout : OceanBase 中未使用，仅兼容作用• partition_names : 将查询限制在某些分区 <p>返回值： 记录列表，每条记录都是一个字典 表示从 column_name 到列值的映射。</p>	<pre>table = self.client.load_table(c ollection_name=test_collectio n_name) where_clause = [table.c["id"] < 100] res = self.client.query(collection_name=test_collectio n_name, output_fields=["id"], filter=where_clause,)</pre>
<pre>def get(self, collection_name: str, ids: Union[list, str, int], output_fields: Optional[List [str]] = None, timeout: Optional[float] = None, partition_names: Optional[List [str]] = None, **kwargs,) -> List[dict]</pre>	<p>获取指定主键 ids 的记录：</p> <ul style="list-style-type: none">• collection_name: 表名称• ids: 某个 id 或者一组 id 列表• output_fields: 输出列（或称为投影列）• timeout : OceanBase 中未使用，仅兼容作用• partition_names : 将查询限制在某些分区 <p>返回值： 记录列表，每条记录都是一个字典 表示从 column_name 到列值的映射。</p>	<pre>res = self.client.get(collection_name=test_collectio n_name, output_fields=["id", "meta"], ids=[80, 12, 112],)</pre>

```
def delete(
self,
collection_name: str,
ids: Optional[Union[list, str,
int]] = None,
timeout: Optional[float] =
None, # pylint:
disable=unused-argument
filter=None,
partition_name: Optional[str]
= "",
**kwargs, # pylint:
disable=unused-argument
)
```

删除集合中的数据

- collection_name: 表名称
- ids: 某个 id 或者一组 id 列表
- timeout : OceanBase 中未使用，仅兼容作用
- filter : 使用带过滤条件的向量近似邻近搜索
- partition_name : 将删除操作限制在某个分区

```
self.client.delete(
collection_name=test_collectio
n_name, ids=[12, 112],
partition_name="p0"
)
```

<pre>def insert(self, collection_name: str, data: Union[Dict, List[Dict]], timeout: Optional[float] = None, partition_name: Optional[str] = "")</pre>	<p>向表中插入数据</p> <ul style="list-style-type: none"> • collection_name: 表名称 • data: 以 Key-Value 形式描述的待插入数据 • timeout : OceanBase 中未使用，仅兼容作用 • partition_name : 将插入操作限制在某个分区 	<pre>data = [{"id": 12, "embedding": [1, 2, 3], "meta": {"doc": "oceanbase document 1"}}, { "id": 90, "embedding": [0.13, 0.123, 1.213], "meta": {"doc": "oceanbase document 1"}, }, {"id": 112, "embedding": [1, 2, 3], "meta": None}, {"id": 190, "embedding": [0.13, 0.123, 1.213], "meta": None},] self.client.insert(collection_name=test_collection_name, data=data)</pre>
<pre>def upsert(self, collection_name: str, data: Union[Dict, List[Dict]], timeout: Optional[float] = None, # pylint: disable=unused-argument partition_name: Optional[str] = "",) -> List[Union[str, int]]</pre>	<p>更新表中的数据。如果主键重复，则替换它</p> <ul style="list-style-type: none"> • collection_name: 表名称 • data: 待更新插入的数据，格式与 insert 接口一致 • timeout : OceanBase 中未使用，仅兼容作用 • partition_name : 将插入操作限制在某个分区 	<pre>data = [{"id": 112, "embedding": [1, 2, 3], "meta": {"doc":'hhh1'}}, {"id": 190, "embedding": [0.13, 0.123, 1.213], "meta": {'doc':'hhh2'}},] self.client.upsert(collection_name=test_collection_name, data=data)</pre>

<pre>def perform_raw_text_sql (self, text_sql: str): return super(). perform_raw_text_sql (text_sql)</pre>	<p>直接执行 SQL 语句</p> <ul style="list-style-type: none">• text_sql: 待执行的 SQL <p>返回值:</p> <p>返回 SQLAlchemy 提供的结果集合迭代器</p>	
--	---	--

13.2 ObVecClient

13.2.5 构造函数

```
def __init__(
self,
uri: str = "127.0.0.1:2881",
user: str = "root@test",
password: str = "",
db_name: str = "test",
**kwargs,
)
```

13.2.6 表模式相关操作

API 接口	参数描述	示例或备注
<pre>def check_table_exists(self, table_name: str)</pre>	<p>检查表是否存在</p> <ul style="list-style-type: none">• table_name: 表名称	

<pre>def create_table(self, table_name: str, columns: List[Column], indexes: Optional[List[Index]] = None, partitions: Optional [ObPartition] = None,)</pre>	<p>创建表</p> <ul style="list-style-type: none"> • table_name: 表名称 • columns: 使用 SQLAlchemy 定义的表的列模式 • indexes: 使用 SQLAlchemy 定义的一组索引表模式 • partitions: 可选的分区规则（详见使用 ObPartition 定义分区规则小节） 	
<pre>@classmethod def prepare_index_params (cls)</pre>	<p>创建一个 IndexParams 对象来记录向量索引表的模式定义</p> <pre>class IndexParams: """Vector index parameters for MilvusLikeClient" def init(self): self._indexes = {}</pre> <p>IndexParams 的定义非常简单，内部只有一个字典类型的成员存放了（列名，索引名）的 tuple 到 IndexParam 结构的映射 IndexParam 类的构造函数为</p> <pre>def init(self, index_name: str, field_name: str, index_type: Union [VecIndexType, str], **kwargs)</pre>	<p>给出一个创建向量索引的使用案例：</p>

- index_name: 向量索引表名
- field_name: 向量列名
- index_type: 向量索引算法类型的一个枚举类, 目前仅支持 HNSW

通过 `prepare_index_params` 获得一个 `IndexParams` 后, 可以通过 `add_index` 接口来注册 `IndexParam`:

```
def add_index(  
    self,  
    field_name: str,  
    index_type: VecIndexType,  
    index_name: str,  
    **kwargs  
)
```

参数含义通 `IndexParam` 的构造函数

```
idx_params = self.client.  
prepare_index_params()  
idx_params.add_index(  
    field_name="title_vector",  
    index_type="HNSW",  
    index_name="  
vidx_title_vector",  
    metric_type="L2",  
    params={"M": 16,  
            "efConstruction": 256},  
)  
self.client.create_collection(  
    collection_name=test_collectio  
n_name,  
    schema=schema,  
  
    index_params=idx_params,  
)
```

需要注意的是

`prepare_index_params` 函数建议在 `MilvusLikeClient` 下使用, 不建议在 `ObVecClient` 中使用, `ObVecClient` 模式下应使用 `create_index` 接口来定义向量索引表。(详见 `create_index` 接口)

<pre>def create_table_with_index_params(self, table_name: str, columns: List[Column], indexes: Optional[List[Index]] = None, vidxs: Optional[IndexParams] = None, partitions: Optional[ObPartition] = None,)</pre>	<p>使用可选的 <code>index_params</code> 在创建表的同时创建向量索引</p> <ul style="list-style-type: none"> • <code>table_name</code>: 表名称 • <code>columns</code>: 使用 SQLAlchemy 定义的表的列模式 • <code>indexes</code>: 使用 SQLAlchemy 定义的一组索引表模式 • <code>vidxs</code>: 通过 <code>IndexParams</code> 指定的向量索引表模式 • <code>partitions</code>: 可选的分区规则（详见使用 <code>ObPartition</code> 定义分区规则章节） 	<p>建议在 <code>MilvusLikeClient</code> 下使用，不建议在 <code>ObVecClient</code> 中使用</p>
<pre>def create_index(self, table_name: str, is_vec_index: bool, index_name: str, column_names: List[str], vidx_params: Optional[str] = None, **kw,)</pre>	<p>支持创建普通索引和向量索引两种模式</p> <ul style="list-style-type: none"> • <code>table_name</code>: 表名称 • <code>is_vec_index</code>: 是索引或是向量索引 • <code>index_name</code>: 索引名称 • <code>column_names</code>: 在哪些列上创建索引 • <code>vidx_params</code>: 向量索引参数，例如 <code>"distance=l2, type=hnslib=vsag"</code> <p>目前 OceanBase 仅支持 <code>type=hnslib=vsag</code> 以及 <code>lib=vsag</code> 请保留这两个的设置，<code>distance</code> 可以设置为 <code>l2</code> 或者 <code>inner_product</code></p>	<pre>self.client.create_index(test_collection_name, is_vec_index=True, index_name="vidx", column_names= ["embedding"], vidx_params="distance=l2, type=hnslib=vsag",)</pre>

<pre>def create_vidx_with_vec_index_param(self, table_name: str, vidx_param: IndexParam,)</pre>	<p>使用向量索引参数创建向量索引</p> <ul style="list-style-type: none"> • table_name: 表名称 • vidx_param: IndexParam 构造的向量索引参数 	
<pre>def drop_table_if_exist(self, table_name: str)</pre>	<p>删除表</p> <ul style="list-style-type: none"> • table_name: 表名称 	
<pre>def drop_index(self, table_name: str, index_name: str)</pre>	<p>删除索引</p> <ul style="list-style-type: none"> • table_name: 表名称 • index_name: 索引名称 	
<pre>def refresh_index(self, table_name: str, index_name: str, trigger_threshold: int = 10000,)</pre>	<p>刷新向量索引表以提升读取性能，可以理解为对增量数据的搬迁</p> <ul style="list-style-type: none"> • table_name: 表名称 • index_name: 索引名称 • trigger_threshold: 刷新动作的触发阈值，如果索引表数据量超过该阈值，则进行刷新 	
<pre>def rebuild_index(self, table_name: str, index_name: str, trigger_threshold: float = 0.2,)</pre>	<p>重建向量索引表以提升读取性能，可以理解为将增量数据合并入基线索引数据</p> <ul style="list-style-type: none"> • table_name: 表名称 • index_name: 索引名称 • trigger_threshold: 重建动作的触发阈值，值域 0 到 1，增量数据占全量数据比例达到该阈值时触发重建 	

13.2.7 DML 操作

API 接口	参数描述	示例或备注
--------	------	-------

<pre>def insert(self, table_name: str, data: Union[Dict, List[Dict]], partition_name: Optional[str] = "",)</pre>	<p>向表中插入数据</p> <ul style="list-style-type: none">• table_name: 表名称• data: 以 Key-Value 形式描述的待插入数据• partition_name: 将插入操作限制在某个分区	<pre>vector_value1 = [0.748479, 0.276979, 0.555195] vector_value2 = [0, 0, 0] data1 = [{"id": i, "embedding": vector_value1} for i in range (10)] data1.extend([{"id": i, "embedding": vector_value2} for i in range(10, 13)]) data1.extend([{"id": i, "embedding": vector_value2} for i in range(111, 113)]) self.client.insert (test_collection_name, data=data1)</pre>
<pre>def upsert(self, table_name: str, data: Union[Dict, List[Dict]], partition_name: Optional[str] = "",)</pre>	<p>更新表中的数据。如果主键重复，则替换它。</p> <ul style="list-style-type: none">• table_name: 表名称• data: 待更新插入数据，key-value 格式• partition_name: 将更新插入限制在某些分区	

<pre>def update(self, table_name: str, values_clause, where_clause=None, partition_name: Optional[str] = "",)</pre>	<p>更新表中的数据。如果主键重复，则替换它。</p> <ul style="list-style-type: none"> • table_name: 表名称 • values_clause: 更新列的值 • where_clause: 更新条件 • partition_name: 将更新操作限制在某些分区 	<pre>data = [{"id": 112, "embedding": [1, 2, 3], "meta": {'doc': 'hhh1'}}, {"id": 190, "embedding": [0.13, 0.123, 1.213], "meta": {'doc': 'hhh2'}},] client.insert(collection_name=test_collection_name, data=data) client.update(table_name=test_collection_name, values_clause=[{'meta': {'doc': 'HHH'}}], where_clause=[text("id=112")])</pre>
<pre>def delete(self, table_name: str, ids: Optional[Union[list, str, int]] = None, where_clause=None, partition_name: Optional[str] = "",)</pre>	<p>删除表中的数据</p> <ul style="list-style-type: none"> • table_name: 表名称 • ids: 某个 id 或者一组 id 列表 • where_clause: 删除条件 • partition_name: 将删除操作限制在某些分区 	<pre>self.client.delete (test_collection_name, ids= ["bcd", "def"])</pre>

<pre>def get(self, table_name: str, ids: Optional[Union[list, str, int]], where_clause = None, output_column_name: Optional[List[str]] = None, partition_names: Optional[List [str]] = None,)</pre>	<p>获取指定主键 <code>ids</code> 的记录。</p> <ul style="list-style-type: none"> • <code>table_name</code>: 表名称 • <code>ids</code>: 某个 id 或者一组 id 列表 • <code>where_clause</code>: 获取条件 • <code>output_column_name</code>: 一组输出列或者投影列名称 • <code>partition_names</code>: 将获取操作限制在某些分区 <p>返回值: 不同于 <code>MilvusLikeClient</code>, <code>ObVecClient</code> 的返回值为一个 tuple list, 每个 tuple 代表一行记录</p>	<pre>res = self.client.get(test_collection_name, ids=["abc", "bcd", "cde", "def"], where_clause=[text("meta->'\$. page' > 1")], output_column_name=['id'])</pre>
<pre>def set_ob_hnsw_ef_search (self, ob_hnsw_ef_search: int)</pre>	<p>设置 HNSW 索引的 <code>efSearch</code> 参数。<code>session</code> 级别变量设置, <code>ef_search</code> 越大召回率越高, 但是查询性能有所下降。</p> <ul style="list-style-type: none"> • <code>ob_hnsw_ef_search</code>: HNSW 索引的 <code>efSearch</code> 参数 	
<pre>def get_ob_hnsw_ef_search (self) -> int</pre>	<p>获取 HNSW 索引的 <code>efSearch</code> 参数</p>	

<pre>def ann_search(self, table_name: str, vec_data: list, vec_column_name: str, distance_func, with_dist: bool = False, topk: int = 10, output_column_names: Optional[List[str]] = None, extra_output_cols: Optional [List] = None, where_clause=None, partition_names: Optional[List [str]] = None, **kwargs,)</pre>	<p>执行向量近似邻近搜索</p> <ul style="list-style-type: none"> • table_name: 表名称 • vec_data: 需要搜索的向量数据 • vec_column_name: 需要搜索的向量列名 • distance_func: 距离函数。提供了 SQLAlchemy func 的扩展, 可选值有 <code>func.l2_distance</code>/<code>func.cosine_distance</code>/<code>func.inner_product</code>/<code>func.negative_inner_product</code>, 分别表示 l2 距离函数、cosine 距离函数、内积距离函数、内积距离的负值 • with_dist: 是否返回带向量距离的结果 • topk: 取最近多少个向量 • output_column_names: 一组输出列或者投影列名称 • extra_output_cols: 额外输出列, 可以提供更复杂的输出表达式 • where_clause: 过滤条件 • partition_names: 将查询限制在某些分区 <p>返回值: 不同于 <code>MilvusLikeClient</code>, <code>ObVecClient</code> 的返回值为一个 tuple list, 每个 tuple 代表一行记录</p>	<pre>res = self.client.ann_search(test_collection_name, vec_data=[0, 0, 0], vec_column_name=" embedding", distance_func=func. l2_distance, with_dist=True, topk=5, output_column_names=["id"],)</pre>
--	---	---

<pre>def precise_search(self, table_name: str, vec_data: list, vec_column_name: str, distance_func, topk: int = 10, output_column_names: Optional[List[str]] = None, where_clause=None, **kwargs,)</pre>	<p>执行精确邻近搜索算法</p> <ul style="list-style-type: none"> • table_name: 表名称 • vec_data: 查询的向量 • vec_column_name: 向量列名 • distance_func: 向量距离函数, 提供了 SQLAlchemy func 的扩展, 可选值有 func.l2_distance/func.cosine_distance/func.inner_product/func.negative_inner_product, 分别表示 l2 距离函数、cosine 距离函数、内积距离函数、内积距离的负值 • topk: 取最近多少个向量 • output_column_names: 一组输出列或者投影列名称 • where_clause: 过滤条件 <p>返回值: 不同于 MilvusLikeClient, ObVecClient 的返回值为一个 tuple list, 每个 tuple 代表一行记录</p>	
<pre>def perform_raw_text_sql(self, text_sql: str)</pre>	<p>直接执行 SQL 语句</p> <ul style="list-style-type: none"> • text_sql: 待执行的 SQL <p>返回值: 返回 SQLAlchemy 提供的结果集合迭代器</p>	

13.3 使用 ObPartition 定义分区规则

pyobvector 提供了以下类型来支持 range/range columns、list/list columns、hash、key 以及二级分区:

- ObRangePartition: range 一级分区。构造时设置 `is_range_columns = True` 以创建 range columns 分区。
- ObListPartition: list 一级分区。构造时设置 `is_list_columns = True` 以创建 list columns 分区。
- ObHashPartition: hash 一级分区。
- ObKeyPartition: key 一级分区。

- ObSubRangePartition: 二级 range 分区。构造时设置 `is_range_columns = True` 以创建 range columns 二级分区。
- ObSubListPartition: list 二级分区。构造时设置 `is_list_columns = True` 以创建 list columns 二级分区。
- ObSubHashPartition: hash 二级分区。
- ObSubKeyPartition: key 二级分区。

13.3.8 range 分区示例

```
range_part = ObRangePartition(  
False,  
range_part_infos=[  
RangeListPartInfo("p0", 100),  
RangeListPartInfo("p1", "maxvalue"),  
],  
range_expr="id",  
)
```

13.3.9 list 分区示例

```
list_part = ObListPartition(  
False,  
list_part_infos=[  
RangeListPartInfo("p0", [1, 2, 3]),  
RangeListPartInfo("p1", [5, 6]),  
RangeListPartInfo("p2", "DEFAULT"),  
],  
list_expr="col1",  
)
```

13.3.10 hash 分区示例

```
hash_part = ObHashPartition("col1", part_count=60)
```

13.3.11 多级分区示例

```
# 一级range分区
range_columns_part = ObRangePartition(
    True,
    range_part_infos=[
        RangeListPartInfo("p0", 100),
        RangeListPartInfo("p1", 200),
        RangeListPartInfo("p2", 300),
    ],
    col_name_list=["col1"],
)

# 二级range分区
range_sub_part = ObSubRangePartition(
    False,
    range_part_infos=[
        RangeListPartInfo("mp0", 1000),
        RangeListPartInfo("mp1", 2000),
        RangeListPartInfo("mp2", 3000),
    ],
    range_expr="col3",
)

range_columns_part.add_subpartition(range_sub_part)
```

13.4 纯 SQLAlchemy API 模式

如果你希望在 OceanBase 数据库的向量检索功能下使用纯粹的 SQLAlchemy API，可以通过如下两种方式获取同步的数据库引擎：

- 方式一：使用 ObVecClient 辅助创建数据库引擎


```
from pyobvector import ObVecClient

client = ObVecClient(uri="127.0.0.1:2881", user="test@test")
engine = client.engine
# 接下来正常使用 SQLAlchemy 创建 session, 使用 SQLAlchemy 的 API 即可
```

- 方式二：使用 ObVecClient 的 `create_engine` 接口创建数据库引擎

```
import pyobvector
from sqlalchemy.dialects import registry
from sqlalchemy import create_engine

uri: str = "127.0.0.1:2881"
user: str = "root@test"
password: str = ""
db_name: str = "test"
registry.register("mysql.oceanbase", "pyobvector.schema.dialect",
"OceanBaseDialect")
connection_str = (
# mysql+oceanbase 表示选择 mysql 标准并使用 OceanBase 数据库的同步驱动
f"mysql+oceanbase://{user}:{password}@{uri}/{db_name}?charset=utf8mb4"
)
engine = create_engine(connection_str, **kwargs)
# 接下来正常使用 SQLAlchemy 创建 session, 使用 SQLAlchemy 的 API 即可
```

如果期望使用 SQLAlchemy 的异步接口，可以使用 OceanBase 数据库的异步驱动：

```
import pyobvector
from sqlalchemy.dialects import registry
from sqlalchemy.ext.asyncio import create_async_engine

uri: str = "127.0.0.1:2881"
```

```
user: str = "root@test"
password: str = ""
db_name: str = "test"
registry.register("mysql.oceanbase", "pyobvector", "AsyncOceanBaseDialect")
connection_str = (
    # mysql+aoceanbase 表示选择 mysql 标准并使用 OceanBase 数据库的异步驱动
    f"mysql+aoceanbase://{user}:{password}@{uri}/{db_name}?charset=utf8mb4"
)
engine = create_async_engine(connection_str)
# 接下来正常使用 SQLAlchemy 创建 session，使用 SQLAlchemy 的 API 即可
```

13.5 更多示例

访问 [pyobvector 代码仓库](#) 获取更多示例。

14 向量检索常见问题

本文介绍了向量检索使用过程中可能遇到的一些常见问题及相关的原因和解决方法。

14.1 在什么模式下支持向量检索？

OceanBase 的 MySQL 模式支持向量检索，Oracle 模式暂不支持。

14.2 向量列中，每一行的数据维度是否必须相同？

必须相同，定义向量列时必须指定维度，写入向量数据时也必须校验维度。

14.3 最大支持写入多少行的向量数据？

不限，取决于租户内存资源。

14.4 如何对超过 2000 维的向量建索引？

需要对数据做维度压缩，压缩到 2000 维以内之后再建索引。

14.5 OceanBase 是否会支持内置的 Embedding 方法和内置的模型算法？

暂时不支持，未来会有计划。