上一篇文章 [一篇文章带你学会向量数据库Milvus（一）]我们我们学习了 Milvus 向量数据库的 数据库管理 ，集合管理，schema 管理。这一篇文章我们继续学习向量数据库的其他内容。

索引管理

Milvus 提供多种 索引类型 来对字段值进行排序，以实现高效的相似性搜索。它还提供三种度量类型：**余弦相似度** (COSINE)、**欧几里得距离** (L2) 和**内积** （IP）来测量向量嵌入之间的距离。

> 建议对经常使用的向量字段和标量字段创建索引

如果集合创建请求中指定了以下任一条件， `Milvus` 在创建集合时会自动生成索引并将其加载到内存中：

- 向量的维度和类型
- schema 和索引参数

下面的代码片段重新调整了现有代码的用途，以建立与 Milvus 实例的连接并创建一个集合，而无需指定其索引参数。在这种情况下，集合缺少索引并且保持卸载状态。

```
ini复制代码 from pymilvus import MilvusClient, DataType

 # 实例化客户端，连接 Milvus 服务
 client = MilvusClient(
     uri="http://localhost:19530"
 )

 # 创建 schema
 schema = MilvusClient.create_schema(
     auto_id=False,
```

集合索引

要为集合创建索引或为集合建立索引，我们需要设置索引参数并调用 `create_index()` 。

```
ini复制代码 # 设置索引的参数
 index_params = MilvusClient.prepare_index_params()

 # 在向量字段 vector 上面添加一个索引
```

```
index_params.add_index(
    field_name="vector",
    metric_type="COSINE",
    index_type=,
    index_name="vector_index"
)

# 在集合demo_v4创建索引文件
client.create_index(
    collection_name="demo_v4",
    index_params=index_params
)
```

> Milvus 目前只支持为集合的每个字段创建一个索引文件

查看索引详细信息

创建索引后我们可以检索索引的详细信息：

```
ini复制代码 res = client.list_indexes(
    collection_name="demo_v4"
)

# Output
#
# [
#     "vector_index",
# ]

res = client.describe_index(
    collection_name="demo_v4",
    index_name="vector_index"
)

# Output
#
# {
#     "index_type": ,
#     "metric_type": "COSINE",
#     "field_name": "vector",
#     "index_name": "vector_index"
# }
```

> 我们可以查看针对特定字段创建的索引文件，并统计使用该索引简历的索引行数。类比 关系性数据库的索引。

删除索引

如果不在需要索引，我们可以删除相关的索引

```
ini复制代码 client.drop_index(
     collection_name="demo_v4",
     index_name="vector_index"
 )
```

检索标量字段

什么是标量字段？变量字段就是除 vector 字段，id 字段之外的字段。在 Milvus 中，标量索引用于加速特定非向量字段值的元过滤，类似于传统的数据库索引。

变量索引类型

- `auto-index` Milvus 根据标量字段的数据类型自动决定索引类型。这适用于不需要控制具体索引类型的情况。
- `custom-index` 可以指定明确的索引类型，比如倒排索引。这就提供了对索引的类型的更多选择。

Auto index 自动索引

要使用自动索引，请省略 **index_type** 参数，以便 `Milvus` 可以根据标量字段类型推断索引类型。

例子：

```
ini复制代码 # Auto indexing
 client = MilvusClient(
     uri="http://localhost:19530"
 )
 #准备一个空的IndexParams对象，无需指定任何索引参数。
 index_params = client.create_index_params()

 index_params.add_index(
     field_name="scalar_1", # 要索引的标量字段的名称
     index_type="", # 要创建的索引类型。对于自动索引，请将其留空或省略此参数。
     index_name="default_index" # 要创建的指数名称
 )

 # 在集合中添加索引

```

```
client.create_index(
  collection_name="demo_v4", # 指定集合名称
  index_params=index_params
)
```

自定义索引

如果我们要使用自定义索引，请在 **index_type** 参数中指定特定索引类型。

看下面的例子：

```
ini复制代码 index_params = client.create_index_params() #  准备一个 IndexPara

index_params.add_index(
    field_name="scalar_2", # 标量字段名称
    index_type="INVERTED", # 明确索引类型
    index_name="inverted_index" # 索引的名称
)

client.create_index(
  collection_name="demo_v4", # 将索引添加到集合中
  index_params=index_params
)
```

对于自定义索引，有效值为：

- **INVERTED**：（推荐）倒排索引由术语词典组成，其中包含按字母顺序排序的所有标记化单词。有关详细信息，请参阅标量索引。
- **STL_SORT**：使用标准模板库排序算法对标量字段进行排序。支持布尔和数字字段（例如 INT8、INT16、INT32、INT64、FLOAT、DOUBLE）。
- **Trie**：用于快速前缀搜索和检索的树形数据结构。支持 VARCHAR 字段。

索引检索

使用 **list_indexes()** 方法验证标量索引的创建：

```
ini复制代码 client.list_indexes(
    collection_name="demo_v4"  # 指定集合名称
)

# Output:
# ['default_index','inverted_index']
```

索引限制

目前，标量索引支持 `INT8`、`INT16`、`INT32`、`INT64`、`FLOAT`、`DOUBLE`、`BOOL` 和 `VARCHAR` 数据类型，但不支持 `JSON` 和 `ARRAY` 类型。

数据 CRUD

在 Milvus 集合的上下文中，实体是集合中单个、可识别的实例。它代表特定类别的独特成员，无论是图书馆中的一本书、基因组中的基因还是任何其他可识别的实体。

集合中的实体共享一组通用的属性，称为schema，概述了每个实体必须遵守的结构，包括字段名称、数据类型和任何其他约束。

将实体成功插入集合中要求提供的数据应包含目标集合的所有架构定义字段。此外，仅当您启用了动态字段时，您还可以包含非架构定义的字段。

准备工作

```
ini复制代码 from pymilvus import MilvusClient

 # Milvus 连接数据库
 client = MilvusClient(
     uri="http://localhost:19530"
 )

 # 创建 collection
 client.create_collection(
     collection_name="demo_v5",
     dimension=5,
     metric_type="IP"
 )
```

插入实体

要插入实体，您需要将数据组织到字典列表中，其中每个字典代表一个实体。每个字典都包含与目标集合中的预定义字段和动态字段对应的键。

```
ini复制代码# 3. Insert some data
data=[
    {"id": 0, "vector": [0.3580376395471989, -0.6023495712049978, 0.184140
    {"id": 1, "vector": [0.19886812562848388, 0.06023560599112088, 0.69769
    {"id": 2, "vector": [0.43742130801983836, -0.5597502546264526, 0.64578
    {"id": 3, "vector": [0.3172005263489739, 0.9719044792798428, -0.369811
    {"id": 4, "vector": [0.4452349528804562, -0.8757026943054742, 0.822077
    {"id": 5, "vector": [0.985825131989184, -0.8144651566660419, 0.6299267
    {"id": 6, "vector": [0.8371977790571115, -0.015764369584852833, -0.310
```

```
    {"id": 7, "vector": [-0.33445148015177995, -0.2567135004164067, 0.8987
    {"id": 8, "vector": [0.39524717779832685, 0.4000257286739164, -0.58905
    {"id": 9, "vector": [0.5718280481994695, 0.24070317428066512, -0.37379
]

res = client.insert(
    collection_name="demo_v5",
    data=data
)
```

插入分区

要将数据插入到特定分区，可以在插入请求中指定分区名称，如下所示：

```
ini复制代码data=[
    {"id": 10, "vector": [-0.5570353903748935, -0.8997887893201304, -0.712
    {"id": 11, "vector": [0.6319019033373907, 0.6821488267878275, 0.855230
    {"id": 12, "vector": [0.9483947484855766, -0.32294203351925344, 0.9759
    {"id": 13, "vector": [-0.5449109892498731, 0.043511240563786524, -0.25
    {"id": 14, "vector": [0.6603339372951424, -0.10866551787442225, -0.943
    {"id": 15, "vector": [-0.8825129181091456, -0.9204557711667729, -0.935
    {"id": 16, "vector": [0.6285586391568163, 0.5389064528263487, -0.31633
    {"id": 17, "vector": [-0.20151825016059233, -0.905239387635804, 0.6749
    {"id": 18, "vector": [0.2432286610792349, 0.01785636564206139, -0.6513
    {"id": 19, "vector": [0.055512329053363674, 0.7100266349039421, 0.4956
]

# 创建分区
client.create_partition(
    collection_name="demo_v4",
    partition_name="partitionA"
)

# 分区中插入数据
res = client.insert(
    collection_name="demo_v4",
    data=data,
    partition_name="partitionA"
)
```

更新插入数据

更新插入数据是更新和插入操作的组合。在 Milvus 中，upsert 操作执行数据级操作，根据集合中是否已存在主键来插入或更新实体。具体来说：

- 如果集合中已存在该实体的主键，则现有实体将被覆盖。
- 如果集合中不存在主键，则将插入一个新实体。

```
ini复制代码data=[
    {"id": 0, "vector": [-0.619954382375778, 0.4479436794798608, -0.174938
    {"id": 1, "vector": [0.4762662251462588, -0.6942502138717026, -0.44900
    {"id": 2, "vector": [-0.8864122635045097, 0.9260170474445351, 0.801326
    {"id": 3, "vector": [0.14594326235891586, -0.3775407299900644, -0.3765
    {"id": 4, "vector": [0.4548498669607359, -0.887610217681605, 0.5655081
    {"id": 5, "vector": [0.11755001847051827, -0.7295149788999611, 0.26081
    {"id": 6, "vector": [0.9363032158314308, 0.030699901477745373, 0.83659
    {"id": 7, "vector": [0.0754823906014721, -0.6390658668265143, 0.561051
    {"id": 8, "vector": [-0.3038434006935904, 0.1279149203380523, 0.503958
    {"id": 9, "vector": [-0.7125086947677588, -0.8050968321012257, -0.3260
]

# 插入与更新
res = client.upsert(
    collection_name='demo_v4',
    data=data
)
```

分区更新也是同样的操作：

```
ini复制代码res = client.upsert(
    collection_name="demo_v4",
    data=data,
    partition_name="partitionA" # 指定分区名称
)
```

删除实体

如果不再需要某个实体，您可以将其从集合中删除。 Milvus 提供两种方式供您识别要删除的实体。

- **通过过滤器删除实体。**
- **按 ID 删除实体。**

```
ini复制代码# 按过滤器删除
res = client.delete(
```

```
    collection_name="quick_setup",
    filter="id in [4,5,6]"
)
ini复制代码# 按 id 删除
res = client.delete(
    collection_name="quick_setup",
    ids=[18, 19],
    partition_name="partitionA"
)
```

向量查询

插入数据后，下一步是在 Milvus 中检索集合执行相似性搜索。

Milvus 允许您进行两种类型的搜索，具体取决于集合中向量字段的数量：

- **单向量搜索**：如果您的集合只有一个向量字段，请使用 `search()` 方法查找最相似的实体。此方法将您的查询向量与集合中的现有向量进行比较，并返回最接近匹配的 ID 以及它们之间的距离。或者，它还可以返回结果的向量值和元数据。
- **多向量搜索**：对于具有两个或多个向量场的集合，请使用 `hybrid_search()` 方法。此方法执行多个近似最近邻 (ANN) 搜索请求，并组合结果以在重新排名后返回最相关的匹配项。

多种搜索类型可以满足不同的搜索需求：

- **基本搜索**：包括单向量搜索、批量向量搜索、分区搜索和指定输出字段搜索。
- **过滤搜索**： 应用基于标量字段的过滤条件来细化搜索结果。
- **范围搜索**： 查找距查询向量特定距离范围内的向量。
- **分组搜索**： 根据特定字段对搜索结果进行分组，以确保结果的多样性。

基本搜索：

发送 `search` 请求时，我们可以提供一个或多个表示查询嵌入的向量值以及指示要返回的结果数的 `limit` 值。根据数据和查询向量，我们获得的结果可能会少于 `limit` 个。当 `limit` 大于查询可能匹配向量的数量时，就会发生这种情况。

单向量搜索

单向量搜索是 Milvus 中 `search` 操作的最简单形式，旨在查找与给定查询向量最相似的向量。

要执行单向量搜索，请指定目标集合名称、查询向量和所需的结果数量（`limit`）。此操作返回一个结果集，其中包含最相似的向量、它们的 ID 以及与查询向量的距离。

批量向量搜索

批量向量搜索通过允许在单个请求中搜索多个查询向量来扩展单向量搜索概念。这种类型的搜索非常适合需要为一组查询向量查找相似向量的场景，从而显着减少所需的时间和计算资源。

在批量向量搜索中，您可以在 `data` 字段中包含多个查询向量。系统并行处理这些向量，为每个查询向量返回一个单独的结果集，每个结果集包含在集合中找到的最接近的匹配项。

下面是从两个查询向量中搜索两个不同的最相似实体集的示例：

```ini复制代码
# 批量向量搜索
res = client.search(
    collection_name="demo_v4",
    data=[
        [0.19886812562848388, 0.06023560599112088, 0.6976963061752597, 0.2
        [0.3172005263489739, 0.9719044792798428, -0.36981146090600725, -0.
    ],
    limit=2,
    search_params={"metric_type": "IP", "params": {}}
)

result = json.dumps(res, indent=4)
```

分区搜索

分区搜索将搜索范围缩小到集合的特定子集或分区。这对于有组织的数据集特别有用，其中数据被分段为逻辑或分类部分，从而通过减少要扫描的数据量来实现更快的搜索操作。

要进行分区搜索，只需在搜索请求的 `partition_names` 中包含目标分区的名称即可。这指定 `search` 操作仅考虑指定分区内的向量。

```ini复制代码
res = client.search(
    collection_name="test_collection",
    data=[[0.02174828545444263, 0.058611125483182924, 0.6168633415965343,
    limit=5,
    search_params={"metric_type": "IP", "params": {}},
    partition_names=["partition_1"] # 这里指定搜索的分区
)
```

使用输出字段进行搜索

使用输出字段进行搜索允许您指定搜索结果中应包含匹配向量的哪些属性或字段。

您可以在请求中指定 `output_fields` 以返回包含特定字段的结果。

```
ini复制代码# 输出字段搜索
res = client.search(
    collection_name="test_collection",
    data=[[0.3580376395471989, -0.6023495712049978, 0.18414012509913835, -
    limit=5,
    search_params={"metric_type": "IP", "params": {}},
    output_fields=["color"] # 返回定义的字段
)
```

过滤搜索

筛选搜索将标量筛选器应用于矢量搜索，允许我们根据特定条件优化搜索结果。

例如，要根据字符串模式优化搜索结果，可以使用 **like** 运算符。此运算符通过考虑前缀、中缀和后缀来启用字符串匹配：

- 若要匹配以特定前缀开头的值，请使用语法. 如: **‘like “prefix%”’** 。
- 若要匹配字符串中任意位置包含特定字符序列的值，请使用语法 **‘like “%infix%”’**
- 若要匹配以特定后缀结尾的值，请使用语法: **‘like “%suffix”’** .
- **like** 运算符还可以通过使用下划线 （_） 表示任何单个字符来用于单字符匹配。 **‘like “y_llow”’** .

筛选**颜色**以**绿色**为前缀的结果：

```
ini复制代码# 过滤器搜索
res = client.search(
    collection_name="test_collection",
    data=[[0.3580376395471989, -0.6023495712049978, 0.18414012509913835, -
    limit=5,
    search_params={"metric_type": "IP", "params": {}},
    output_fields=["color"],
    filter='color like "gree%"'
)
```

范围搜索

范围搜索允许查找距查询向量指定距离范围内的向量。

通过设置 `radius` 和可选的 `range_filter`，可以调整搜索的广度以包含与查询向量有些相似的向量，从而提供潜在匹配的更全面的视图。

- `radius`：定义搜索空间的外边界。只有距查询向量在此距离内的向量才被视为潜在匹配。

- `range_filter`：虽然 `radius` 设置搜索的外部限制，但可以选择使用 `range_filter` 来定义内部边界，创建一个距离范围，在该范围内向量必须落下才被视为匹配。

```
ini复制代码# 范围搜索
search_params = {
    "metric_type": "IP",
    "params": {
        "radius": 0.8, # 搜索圆的半径
        "range_filter": 1.0 # 范围过滤器，用于过滤出不在搜索圆内的向量。
    }
}

res = client.search(
    collection_name="test_collection",
    data=[[0.3580376395471989, -0.6023495712049978, 0.18414012509913835, -
    limit=3, # 返回的搜索结果最大数量
    search_params=search_params,
    output_fields=["color"],
)
```

更多搜索方式可以参考 Milvus 官网。

多向量搜索

在 Milvus2.4 版本开始，引入了多想两支持和混合搜索功能。这就意味着用户可以将多个向量字段引入到单个集合中。不同的向量长可以表示不同的方面，不同的 embedding Model 甚至表征同一实体的不同数据模态。这同时意味着扩展了信息的丰富性。有了这个功能我们可以在综合场景使用，比如：根据图片、语言、指纹等各种属性来识别向量中最相似的人。

多向量搜索支持在各种字段上执行搜索请求，并使用重新排名策略的组合结果。

示例：

```
ini复制代码from pymilvus import connections, Collection, FieldSchema, Colle
import random

# 连接 Milvus
connections.connect(
    host="127.0.0.1", # 可以替换为自己的服务地址
    port="19530"
)

# 创建 schema
```

```
fields = [
    FieldSchema(name="film_id", dtype=DataType.INT64, is_primary=True),
    FieldSchema(name="filmVector", dtype=DataType.FLOAT_VECTOR, dim=5), #
    FieldSchema(name="posterVector", dtype=DataType.FLOAT_VECTOR, dim=5)]

schema = CollectionSchema(fields=fields,enable_dynamic_field=False)

# 创建 collection
collection = Collection(name="test_collection", schema=schema)

# 添加索引
index_params = {
    "metric_type": "L2",
    "index_type": "IVF_FLAT",
    "params": {"nlist": 128},
}
# 字段 filmVector 创建索引
collection.create_index("filmVector", index_params)
# 字段 posterVector 创建索引
collection.create_index("posterVector", index_params)

# 向量数据库中插入的实体
entities = []

for _ in range(1000):
    # 构造实体
    film_id = random.randint(1, 1000)
    film_vector = [ random.random() for _ in range(5) ]
    poster_vector = [ random.random() for _ in range(5) ]

    entity = {
        "film_id": film_id,
        "filmVector": film_vector,
        "posterVector": poster_vector
    }
    entities.append(entity)
# 集合中插入实体
collection.insert(entities)
```

创建 AnnSearchRequest 实例

多向量搜索使用 `hybrid_search()` API 在一次调用中执行多个 ANN 搜索请求。每个 `AnnSearchRequest` 代表特定矢量场上的单个搜索请求。

示例创建两个 `AnnSearchRequest` 实例以对两个向量字段执行单独的相似性搜索。

```
makefile复制代码from pymilvus import AnnSearchRequest

# 创建多搜索请求 filmVector
query_filmVector = [[0.8896863042430693, 0.370613100114602, 0.237793150771

search_param_1 = {
    "data": query_filmVector, # 请求查询的向量数据
    "anns_field": "filmVector", # 搜索的向量字段 filmVector
    "param": {
        "metric_type": "L2", # 该参数值必须与集合模式中使用的值相同。
        "params": {"nprobe": 10}
    },
    "limit": 2 # 限定 AnnSearchRequest 搜索中返回的结束数量
}
request_1 = AnnSearchRequest(**search_param_1)

# 创建多搜索请求 posterVector
query_posterVector = [[0.02550758562349764, 0.006085637357292062, 0.532525
search_param_2 = {
    "data": query_posterVector, #请求查询的向量数据
    "anns_field": "posterVector", # 搜索的向量字段 posterVector
    "param": {
        "metric_type": "L2", # 该参数值必须与集合模式中使用的值相同。
        "params": {"nprobe": 10}
    },
    "limit": 2 # 限定 AnnSearchRequest 搜索中返回的结束数量
}
request_2 = AnnSearchRequest(**search_param_2)

reqs = [request_1, request_2]
```

配置排名策略

创建 `AnnSearchRequest` 实例后，配置重新排名策略以组合结果并重新排名。目前有两个选项：`WeightedRanker` 和 `RRFRanker`。

- 使用加权评分： `WeightedRanker` 用于为每个具有指定权重的向量场搜索结果分配重要性。如果将某些向量字段的优先级置于其他向量字段之上， `WeightedRanker(value1, value2, ..., valueN)` 可以在组合搜索结果中提现出来。

```
ini复制代码from pymilvus import WeightedRanker
# 使用 WeightedRanker 来结合具有指定权重的结果
```

```
# 将文本搜索赋予权重 0.8，将图像搜索赋予权重 0.2。
rerank = WeightedRanker(0.8, 0.2)
```

> 使用 `WeightedRanker` 时，请注意：
> 每个权重值的范围从 0（最不重要）到 1（最重要），影响最终的总分。
> `WeightedRanker` 中提供的权重值总数应等于您创建的 `AnnSearchRequest` 实例的数量。

执行混合搜索

设置 `AnnSearchRequest` 实例和重新排名策略后，使用 `hybrid_search()` 方法执行多向量搜索。

```
ini复制代码# 在进行多向量搜索之前，将集合加载到内存中。
collection.load()

res = collection.hybrid_search(
    reqs, # 第1步创建的AnnSearchRequests列表
    rerank, # 在第2步指定的重新排序策略
    limit=2 # 限定最终返回数据量
)
```

参数说明：

- reqs： 搜索请求列表，其中每个请求都是一个 `ANNSearchRequest` 对象。每个请求可以对应于不同的矢量场和不同的搜索参数集。
- `rerank` （*对象*）：用于混合搜索的重新排名策略。可能的值： `WeightedRanker(value1, value2, ..., valueN)` 和 `RRFRanker()` 。
- `limit` (*int*)：混合搜索中返回的最终结果的最大数量。

limits 限制

- 通常，每个集合默认允许最多 4 个向量字段。但是，您可以选择调整 `proxy.maxVectorFieldNum` 配置以扩展集合中矢量字段的最大数量，每个集合的最大限制为 10 个矢量字段
- 集合中部分索引或加载的向量字段将导致错误。
- 目前，混合搜索中的每个 `AnnSearchRequest` 只能携带一个查询向量。

读者福利：如果大家对大模型感兴趣，这套大模型学习资料一定对你有用

对于0基础小白入门：

> 如果你是零基础小白，想快速入门大模型是可以考虑的。
> 一方面是学习时间相对较短，学习内容更全面更集中。
> 二方面是可以根据这些资料规划好学习计划和方向。