

1. 常见向量数据库的介绍

1.0 复习知识

当计算两个向量的相似度时，常见的方法包括余弦相似度、内积和L2距离（欧几里得距离）。下面是每个方法的公式和示例计算。

1. 余弦相似度

公式：

$$\text{cosine similarity}(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|}$$

其中， $\mathbf{a} \cdot \mathbf{b}$ 是向量 \mathbf{a} 和 \mathbf{b} 的内积， $\|\mathbf{a}\|$ 和 $\|\mathbf{b}\|$ 是向量的欧几里得范数（L2范数）。

计算示例：

假设有两个向量：

$$\mathbf{a} = [1, 2, 3], \quad \mathbf{b} = [4, 5, 6]$$

1. 计算内积： $\mathbf{a} \cdot \mathbf{b} = 1 \times 4 + 2 \times 5 + 3 \times 6 = 32$
2. 计算范数： $\|\mathbf{a}\| = \sqrt{1^2 + 2^2 + 3^2} = \sqrt{14}$, $\|\mathbf{b}\| = \sqrt{4^2 + 5^2 + 6^2} = \sqrt{77}$
3. 计算余弦相似度： $\text{cosine similarity}(\mathbf{a}, \mathbf{b}) = \frac{32}{\sqrt{14} \times \sqrt{77}} \approx 0.974$

2. 内积 (Dot Product)

公式：

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^n a_i b_i$$

即向量 \mathbf{a} 和 \mathbf{b} 的内积。

计算示例：

假设有两个向量：

$$\mathbf{a} = [1, 2, 3], \quad \mathbf{b} = [4, 5, 6]$$

1. 计算内积: $\mathbf{a} \cdot \mathbf{b} = 1 \times 4 + 2 \times 5 + 3 \times 6 = 32$

3. L2 距离 (欧几里得距离)

公式:

$$\text{L2 distance}(\mathbf{a}, \mathbf{b}) = \sqrt{\sum_{i=1}^n (a_i - b_i)^2}$$

即计算向量 \mathbf{a} 和 \mathbf{b} 每个元素差值的平方和, 再取平方根。

计算示例:

假设有两个向量:

$$\mathbf{a} = [1, 2, 3], \quad \mathbf{b} = [4, 5, 6]$$

1. 计算每个分量的差值: $(1 - 4)^2 = 9, (2 - 5)^2 = 9, (3 - 6)^2 = 9$
2. 计算平方和: $9 + 9 + 9 = 27$
3. 计算L2距离: $\text{L2 distance}(\mathbf{a}, \mathbf{b}) = \sqrt{27} \approx 5.196$

总结:

- **余弦相似度** 衡量两个向量的夹角相似度, 值范围为 $[-1, 1]$, 值越大表示越相似。
- **内积** 衡量两个向量的相似度, 值越大表示越相似, 但不考虑向量的大小。
- **L2距离** 衡量两个向量之间的“直线”距离, 值越小表示越相似。

1.1 什么是向量数据库

向量数据库是一种专门用于存储和查询高维向量的数据库。随着机器学习和深度学习技术的发展, 向量表示 (如词向量、图像特征、用户行为向量等) 变得越来越重要。这些向量通常是多维的, 表示数据的特征和属性。

在这样的背景下, 向量数据库提供了高效的存储、检索和相似度搜索功能。

向量数据库其实和其他数据库差不多, 只不过他存的是向量。

一般使用向量数据库的流程是：创建数据库->创建集合->创建索引->插入数据->搜索。

1.2 常见的向量数据库

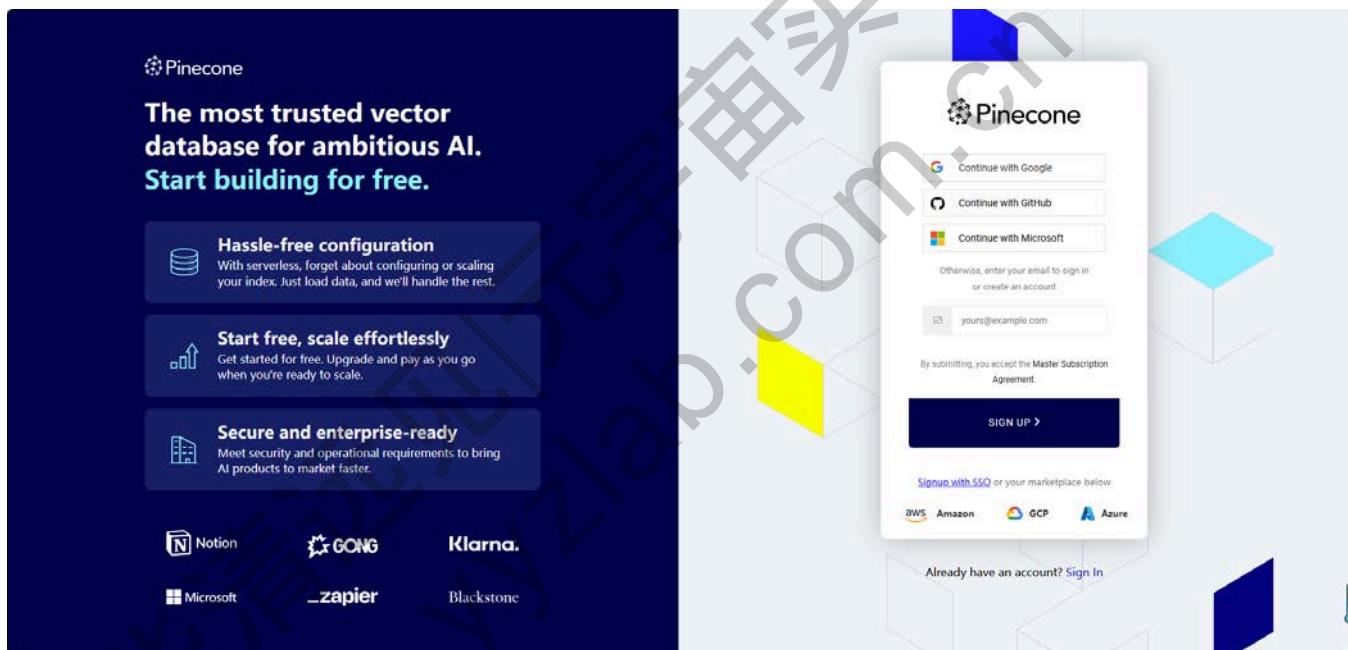
1.2.1 Pinecone

专为构建和运行向量数据库而设计，提供简单易用的API。支持高效的相似性搜索和实时数据更新。提供管理和监控功能，易于扩展。

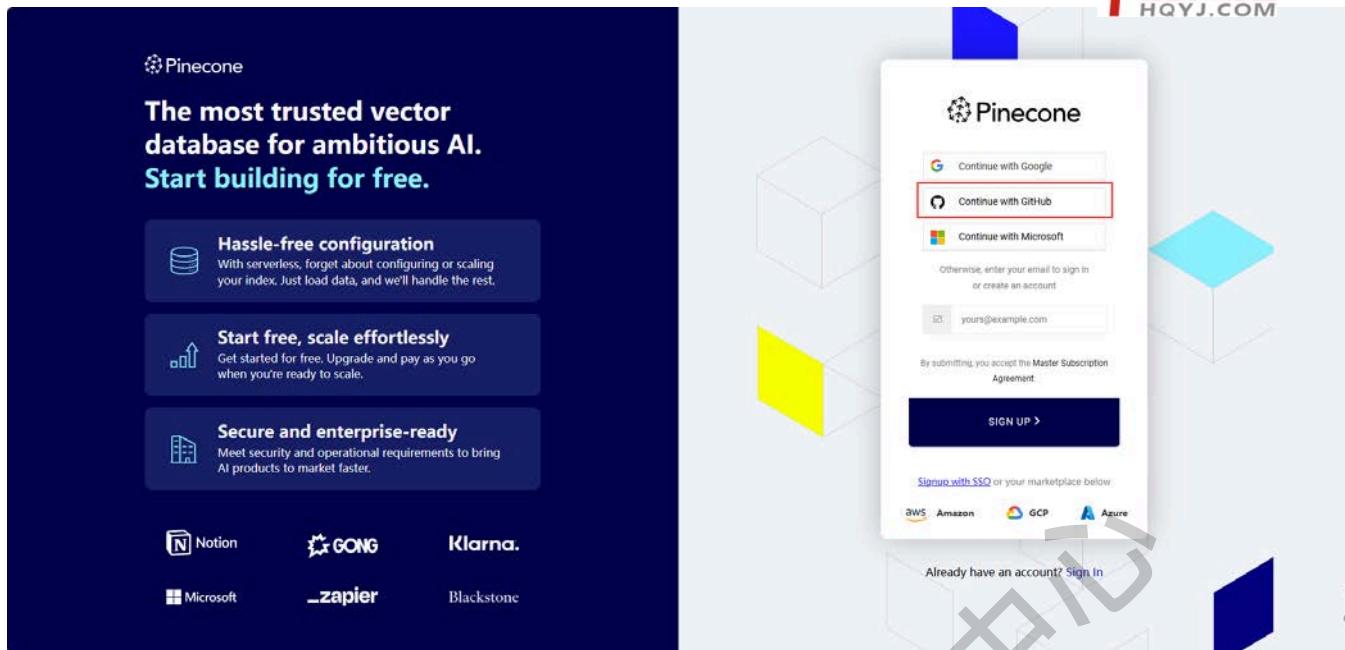
注册Pinecone获取api key

安装 Pinecone

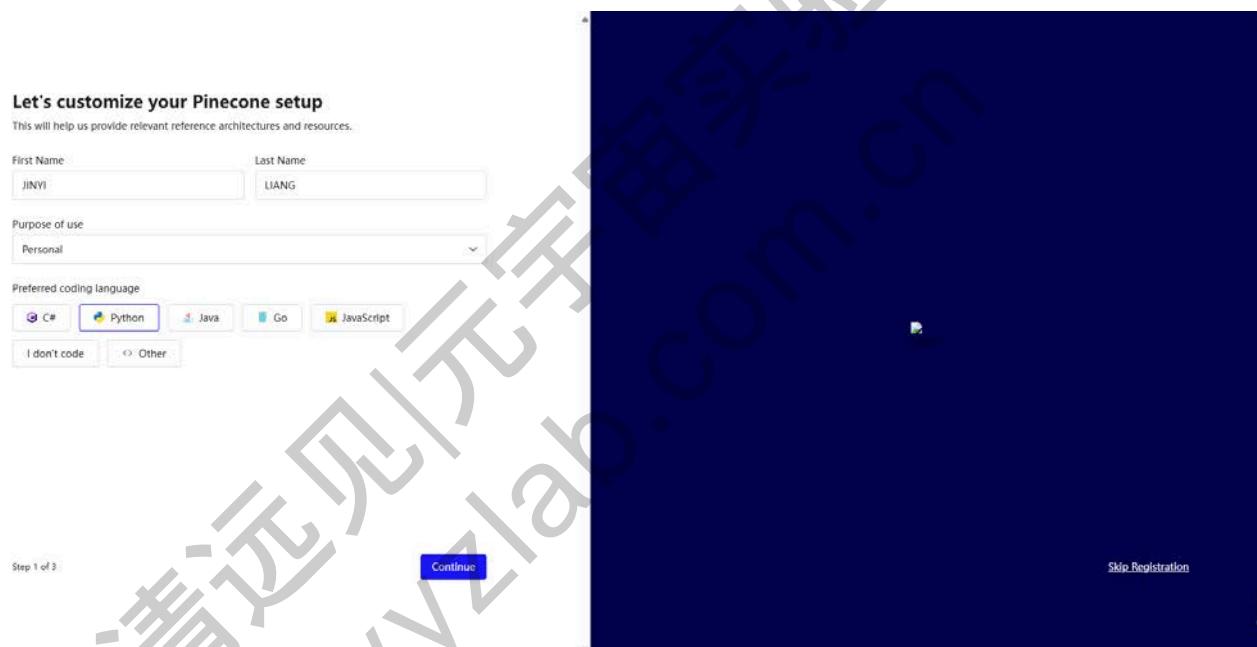
打开 <https://app.pinecone.io>



注册登录，这里直接使用github账号登录



随便填一填信息，后面一样随便填写，然后一直continue



获取api key

The screenshot shows the Pinecone interface for managing API keys. On the left, there's a sidebar with links like 'Get started', 'Database', 'Assistant', 'Inference', 'API keys' (which is selected and highlighted in red), and 'Manage'. Below that is a 'STARTER USAGE' section with RUs, WUs, and Storage metrics, each with a 'Upgrade now' button. The main area is titled 'API keys' and lists one key: 'default' created on '2024/10/14'. The 'Value' column shows a redacted API key. There are 'Actions' buttons for each key.

```
1 | pip install pinecone==5.4.1
```

初始化客户端

```
1 | # 初始化客户端
2 | pc = Pinecone(api_key="4a646aa7-de8d-461e-ac3f-367a37a95bfa")
```

创建 Pinecone 服务端无服务器索引，查询时不注释会报错显示已存在。name为索引库的名字，dimension为加入索引库的向量维度，metric为相似度计算方法这里设置为余弦相似度，spec默认就行。

```
1 | # 创建 Pinecone 服务端无服务器索引（创建时打开注释，查询时注释上。）
2 | index_name = "example"
3 | pc.create_index(
4 |     name=index_name,
5 |     dimension=1024, # Replace with your model dimensions
6 |     metric="cosine", # Replace with your model metric
7 |     spec=ServerlessSpec(
8 |         cloud="aws",
9 |         region="us-east-1"
10 |     )
11 | )
```

向example中加入向量。

```

1 # 向索引中插入向量（创建时打开注释，查询时注释上。）
2 index = pc.Index('example')
3
4 index.upsert(
5     vectors=[
6         {"id": "vec1", "values": embeddings[0]},
7         {"id": "vec2", "values": embeddings[1]},
8         {"id": "vec3", "values": embeddings[2]},
9     ],
10 )

```

查询相似度，vector是要查询的向量，top_k是查询出最接近的3个，include_values为是否显示向量的值。

```

1 # 查询相似向量
2 query_results = index.query(
3     vector=query_embedding[0].tolist(),
4     top_k=3,
5     # include_values=True
6 )
7 print(query_results)
8
9 # {'matches': [{'id': 'vec2', 'score': 0.670338273, 'values': []},
10 #                 {'id': 'vec1', 'score': 0.451030284, 'values': []},
11 #                 {'id': 'vec3', 'score': 0.0896421522, 'values': []}],
12 #     'namespace': '',
13 #     'usage': {'read_units': 5}}

```

如果不需要example索引则删除

```
1 pc.delete_index(index_name)
```

结果解释：

当然，以下是详细的解释以及带有注释的代码：

查询结果解释

查询结果是一个字典，包含以下键：

- `matches`: 匹配的结果列表。
 - 每个匹配项是一个字典，包含以下键：
 - `id`: 向量的唯一标识符。
 - `score`: 相似度分数（余弦相似度）。
 - `values`: 向量的实际值（在本例中为空列表，因为设置了`include_values=False`）。
- `namespace`: 命名空间（默认为空字符串）。
- `usage`: 使用情况统计信息。
 - `read_units`: 读取单元的数量。

具体到你的查询结果：

```

1 {
2   'matches': [
3     {'id': 'vec2', 'score': 0.670338333, 'values': []},
4     {'id': 'vec1', 'score': 0.451030374, 'values': []},
5     {'id': 'vec3', 'score': 0.0896422863, 'values': []}
6   ],
7   'namespace': '',
8   'usage': {'read_units': 5}
9 }
```

- `matches`:
 - 第一个匹配项：{'id': 'vec2', 'score': 0.670338333, 'values': []}
 - `id: 'vec2'` 对应的是“厨房里有一只黑色的猫”。
 - `score: 0.670338333` 表示与查询“黑色的猫在哪？”的相似度最高。
 - 第二个匹配项：{'id': 'vec1', 'score': 0.451030374, 'values': []}
 - `id: 'vec1'` 对应的是“院子里有一只可爱的小猫”。

- `score: 0.451030374` 表示与查询的相似度次之。
- 第三个匹配项: `{'id': 'vec3', 'score': 0.0896422863, 'values': []}`
 - `id: 'vec3'` 对应的是“大模型真简单”。
 - `score: 0.0896422863` 表示与查询的相似度最低。
- **namespace:** 空字符串, 表示没有使用命名空间。
- **usage:**
 - `read_units: 5` 表示本次查询消耗了5个读取单元。

1.2.2 FAISS (Facebook AI Similarity Search)

一个开源库, 专门用于高效的相似性搜索。

支持多种索引结构, 适合处理非常大的数据集。具有强大的性能优化, 能够在CPU和GPU上运行。

安装FAISS

```
1 | pip install faiss-cpu==1.9.0
```

创建索引

创建一个 Faiss 索引, 并将句子的嵌入向量添加进去, 通过`faiss.write_index`可以将向量持久化到本地上。

持久化是指将数据保存到非易失性存储介质上的过程, 以便在程序关闭后仍然能够保留这些数据。

这样在运行代码后, 在你指定的位置会新建一个`faiss_index.index`文件。

```
1 # 向量数据库
2 import faiss
3 import numpy as np
4
5 # 假设我们已经有句子的向量
6 # embeddings 是一个 2D 数组, 形如 (3, 1024) 表示 3 个句子的 1024 维
7 # 向量
8 embeddings = np.array(embeddings)
```

```
9 # 保存索引到磁盘文件
10 faiss.write_index(index, "db/faiss_index.index")
11
12 # 初始化 Faiss 索引
13 index = faiss.IndexFlatIP(1024) # 使用内积进行检索（标准化后的内积
就是余弦相似度）
14 # index = faiss.IndexFlatL2(1024) # 使用L2距离进行检索
15
16 index.add(embeddings) # 添加向量到索引
```

标准化后的内积就是余弦相似度 如何理解：

`faiss.IndexFlatIP` 使用内积进行相似度计算，其背后的原因是，内积在某些情况下可以与余弦相似度等价。如果你将向量进行标准化，即每个向量除以其欧几里得范数，那么内积结果就相当于计算了它们之间的余弦相似度。也就是说，标准化后的内积和余弦相似度是相等的。

标准化后的内积等于余弦相似度

假设我们有两个向量 **a** 和 **b**，并且对它们进行了标准化：

$$\hat{\mathbf{a}} = \frac{\mathbf{a}}{\|\mathbf{a}\|}, \quad \hat{\mathbf{b}} = \frac{\mathbf{b}}{\|\mathbf{b}\|}$$

那么它们之间的内积就是：

$$\hat{\mathbf{a}} \cdot \hat{\mathbf{b}} = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|}$$

这个结果正是余弦相似度的公式。因此，如果在实际应用中，向量已经被标准化，使用内积进行检索实际上就是在计算余弦相似度。

查询并检索

```
1 # 假设我们想查询 "黑色的猫在哪？"
2 query_embedding = model.encode(["黑色的猫在哪？"])
3
4 # 进行检索，返回最相似的 2 个结果
5 D, I = index.search(query_embedding, k=3)
6
7 # 输出结果
8 print(f"最相似的句子索引: {I}")
9 print(f"对应的距离: {D}")
10
11 # 最相似的句子索引: [[1 0 2]]
12 # 对应的距离: [[0.6703382 0.45103022 0.08964214]]
```

1.2.3 Chroma

提供易于使用的API，特别适合构建和使用向量数据库。支持多种向量存储后端，具有良好的扩展性。适用于文本、图像等多种类型的数据。

安装Chroma

```
1 pip install chromadb==0.5.20
```

创建 Chroma 客户端

创建一个 Chroma 客户端实例，并选择一个持久化的存储路径（可选）。如果没有指定路径，默认情况下 Chroma 将在内存中运行。

这样在运行代码后，在你指定的位置会新建一个chroma.sqlite3文件。

```
1 from chromadb import Client, PersistentClient
2 from chromadb.config import Settings
3
4 # 初始化 Chroma 客户端
5 # client = Client()
6 # 持久化
7 client = PersistentClient(path="db")
```

创建或连接到一个集合

metadata={"hnsw:space": "cosine"}表示使用余弦相似度，还可以使用"L2"距离、"ip"内积，注意在Chroma中他们的公式如下：

Distance	parameter	Equation
Squared L2	l2	$d = \sum (A_i - B_i)^2$
Inner product	ip	$d = 1.0 - \sum (A_i \times B_i)$
Cosine similarity	cosine	$d = 1.0 - \frac{\sum(A_i \times B_i)}{\sqrt{\sum(A_i^2)} \cdot \sqrt{\sum(B_i^2)}}$

因此默认返回的是一个cosine，返回值越大两个向量越接近，对于余弦相似度计算，使用1-d可以还原我们更直观的相似度。

```
1 # 创建集合
2 collection = client.create_collection("example_collection",
   metadata={"hnsw:space": "cosine"},)
```

插入数据

```
1 # 插入向量
2 collection.add(
3     embeddings=embeddings,
4     ids=["vec0", "vec1", "vec2"],
5 )
```

查询并检索

```
1 # 查询
2 results = collection.query(
3     query_embeddings=query_embedding,
4     n_results=3
5 )
6
7 # 输出结果
8 print(f"最相似的句子索引: {results['ids']}")  

9 print(f"对应的距离: {[1-d for d in results['distances'][0]]}")  

10
11 # 最相似的句子索引: [['vec1', 'vec0', 'vec2']]
12 # 对应的相似度: [0.6703380942344666, 0.45103007555007935,  

    0.08964204788208008]
```

1.2.4 Milvus

Milvus 是一款云原生向量数据库，它具备高可用、高性能、易拓展的特点，用于海量向量数据的实时召回。

安装 Milvus

设置工作目录和配置文件，在创建启动容器之前，我们要先设置好工作目录和配置文件。

新建 milvus 后在 milvus 文件夹下新建以下文件夹。

名称	修改日期	类型	大小
conf	2024/10/15 11:58	文件夹	
db	2024/10/15 11:58	文件夹	
logs	2024/10/15 11:58	文件夹	
pic	2024/10/15 11:59	文件夹	
volumes	2024/10/15 13:14	文件夹	
wal	2024/10/15 11:59	文件夹	
docker-compose.yml	2024/10/15 11:46	YML 文件	2 KB

下载 docker-compose.yml , v2.4.13是版本。将 docker-compose.yml 复制到刚刚新建的 milvus 文件夹中。

```
1 wget https://github.com/milvus-
io/milvus/releases/download/v2.4.13/milvus-standalone-docker-
compose.yml -O docker-compose.yml
```

使用 Docker 部署 Milvus , 在 milvus 文件夹中打开终端, 运行以下命令

```
1 docker-compose up -d
```

```
infusion"
[+] Running 22/22
✓ standalone Pulled
  ✓ 7646c8da3324 Pull complete   124.9s
  ✓ c3b583ec8f4a Pull complete   15.7s
  ✓ 151cedceed8 Pull complete   39.2s
  ✓ 0290054277a5 Pull complete   66.9s
  ✓ a8af11465778 Pull complete   73.8s
  ✓ 051f2a3eed93 Pull complete   109.7s
  ✓ etcd Pulled                116.2s
  ✓ dbba69284b27 Pull complete   26.7s
  ✓ 270b322b3c62 Pull complete   15.3s
  ✓ 7c21e2da1038 Pull complete   16.4s
  ✓ cb4f77bfee6c Pull complete   17.6s
  ✓ e5485096ca5d Pull complete   19.1s
  ✓ 3ea3736f61e1 Pull complete   20.0s
  ✓ 1e815a2c4f55 Pull complete   20.6s
  ✓ minio Pulled                21.2s
  ✓ c7e856e03741 Pull complete   85.3s
  ✓ c1ff217ec952 Pull complete   45.2s
  ✓ b12cc8972a67 Pull complete   49.7s
  ✓ 4324e307ea00 Pull complete   53.5s
  ✓ 152089595ebc Pull complete   58.6s
  ✓ 05f217fb8612 Pull complete   66.2s
[+] Running 4/4
✓ Network milvus           Created          0.3s
✓ Container milvus-minio    Started         42.1s
✓ Container milvus-etcd     Started         42.1s
```

设置端口 docker port milvus-standalone 19530/tcp。

从下面信息可以看出, 这个Docker环境中四个正在运行的容器, 分别与Milvus数据库服务、Etcd键值存储服务、MinIO对象存储服务相关联。“standalone”指的是一个独立运行的Milvus实例。

Docker desktop

Containers [Give feedback](#)

Container CPU usage: 7.44% / 2000% (20 CPUs available)

Container memory usage: 408.37MB / 14.29GB

Show charts

Search

Only show running containers

Name	Image	Status	Port(s)	CPU (%)	Last started	Actions
hqi_j_ai	hqi_ai	Exited (137)	8889:8888	0%	4 days ago	▶ ⋮ ☰
milvus Running (3/3)						
minio	minio/minio:RELEASE	Running	9000:9000 Show all ports (2)	0%	9 minutes ago	▶ ⋮ ☰
etcd	guay.io/coreos/etc	Running		0.77%	9 minutes ago	▶ ⋮ ☰
standalon	milvusdb/milvus:v	Running	19530:19530 Show all ports (2)	5.75%	9 minutes ago	▶ ⋮ ☰

Showing 5 items

Engine running RAM 2.20 GB CPU 0.55% Not signed in New version available

安装可视化界面（可选）

进入官网 <https://github.com/zilliztech/attu/releases> 下载安装。

Sep 3

namieczz

v2.4.8

4c63e98

Compare

Release v2.4.8 [Latest](#)

What's Changed

- support display segment level
- support create index on primary field
- support setup database properties
- change default drop_ratio_build value to 0.32
- display partition id on the partition page
- bug fix

Full Changelog: [v2.4.7...v2.4.8](#)

Assets 6

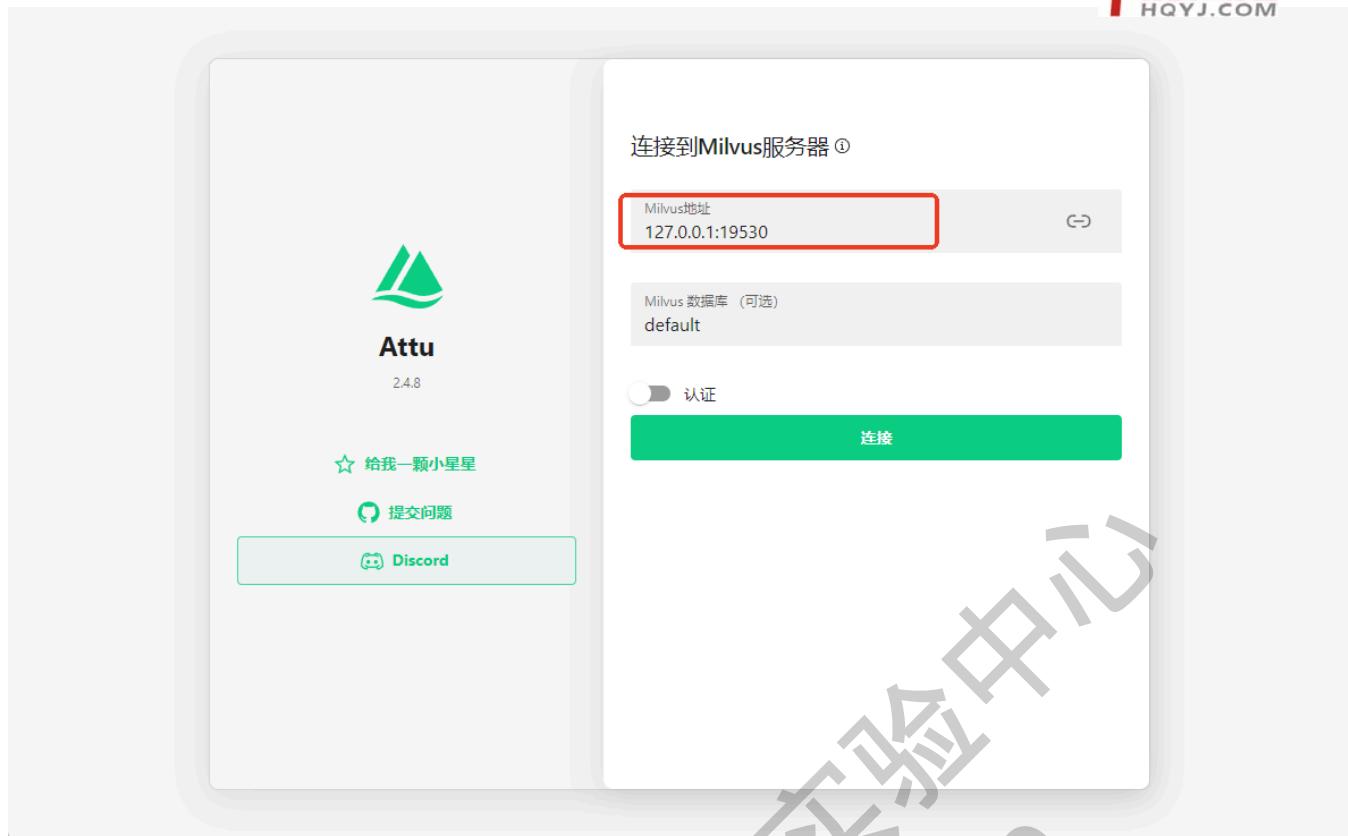
Attu-mac-arm64-v2.4.8.dmg	98.1 MB	Sep 3
Attu-mac-x64-v2.4.8.dmg	103 MB	Sep 3
Attu-Setup-2.4.8.exe	77.4 MB	Sep 3
Attu_2.4.8_dmg64.deb	73.5 MB	Sep 3
Source code (zip)		Sep 3
Source code (tar.gz)		Sep 3

Aug 19

3 people reacted

Release v2.4.7

安装docker的端口填写



连接进入

The screenshot shows the Milvus UI. At the top, it says '欢迎来到 Milvus!' (Welcome to Milvus!). On the left, there's a sidebar with icons for database, collection, user, and cluster. The '数据库' (Database) section shows a 'default' database with '所有Collection' (All Collections) count at 0, '创建时间' (Creation Time) at 2024/10/15 13:14:46, and a '创建数据库' (Create Database) button. The main area has a green banner saying '成功连接到milvus' (Successfully connected to milvus). Below it, '系统信息' (System Information) includes: 'v2.4.13 Milvus Version', 'STANDALONE 部署模式' (Deployment Mode), '18.51 分钟 运行时间' (Run Time), '1 用户' (1 User), and '2 角色' (2 Roles). A small note at the bottom left says 'v 2.4.8'.

通过python使用Milvus

安装

```
1 | pip install pymilvus==2.5.0
```

创建数据库

通过指定的主机地址 (host) 和端口号 (port) 建立与 Milvus 服务的连接。sample_db 为自定义的数据库名字。

```
1 | from pymilvus import connections, db
2 |
3 | conn = connections.connect(host="127.0.0.1", port=19530)
4 | database = db.create_database("sample_db")
```

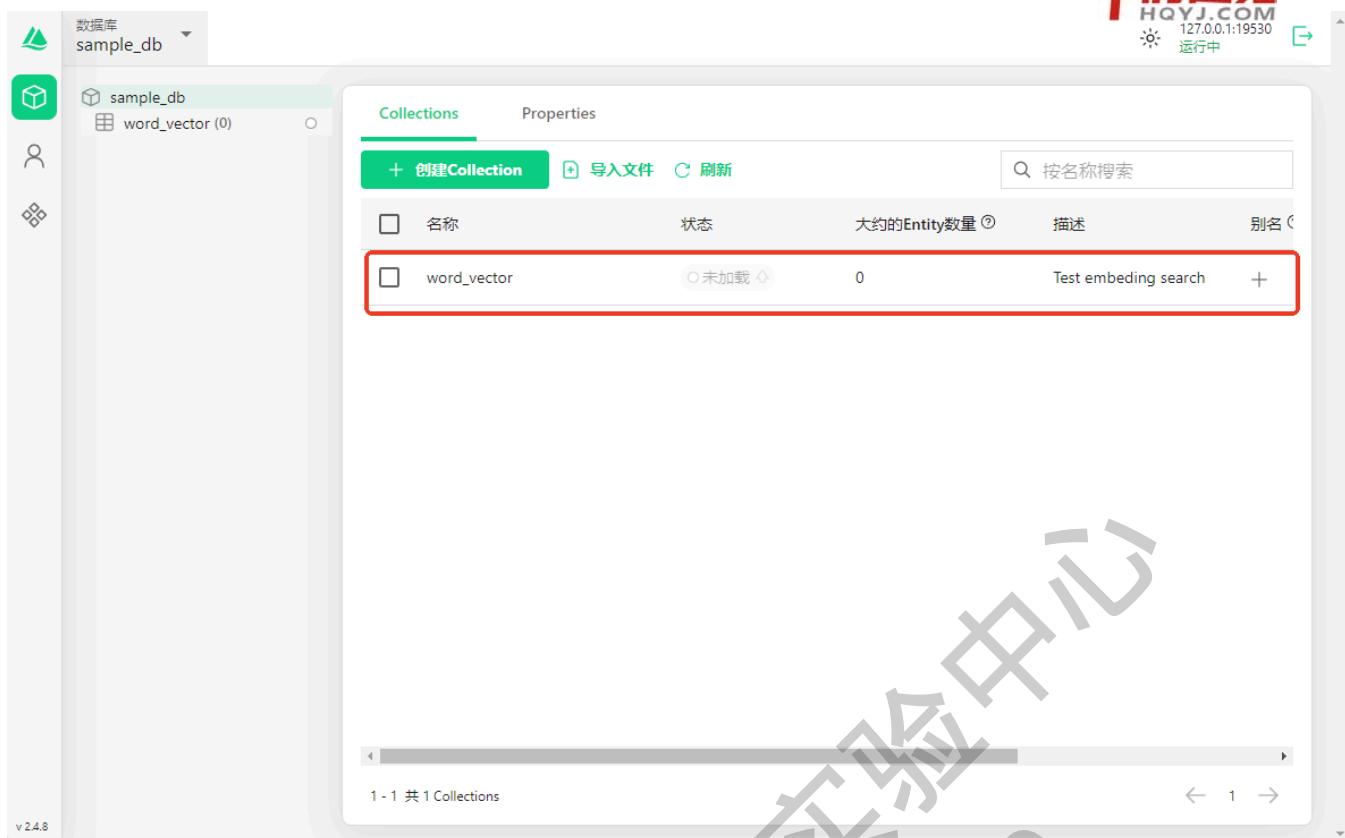


创建collection (集合)

建立连接，创建两个字段id和embedding，name设置名字，dtype设置类型，is_primary表示设置为主键（和SQL类似），auto_id表示自动分配id。dim表示存入的向量维度。

```
1 | # 创建 collection
2 | # 导入必要的类和模块
```

```
3 from pymilvus import CollectionSchema, FieldSchema, DataType
4 from pymilvus import Collection, db, connections
5
6 # 连接到 Milvus 服务
7 conn = connections.connect(host="127.0.0.1", port=19530)
8 # 使用名为 sample_db 的数据库
9 db.using_database("sample_db")
10
11 # 定义主键字段 id, 类型为 INT64, 并设置为主键 (is_primary=True), 同时
12 # 开启自动增长 (auto_id=True)
12 id = FieldSchema(name="id", dtype=DataType.INT64,
13                   is_primary=True, auto_id=True)
13
14 # 定义向量字段 embedding, 类型为 FLOAT_VECTOR, 维度为 1024
15 embedding = FieldSchema(name="embedding",
16                          dtype=DataType.FLOAT_VECTOR, dim=1024)
16
17 # 创建集合模式 (CollectionSchema), 包含 id 和 embedding 字段, 并提供
18 # 描述信息
18 schema = CollectionSchema(
19     fields=[id, embedding],
20     description="Test embedding search",
21 )
22
23 # 定义集合名称为 word_vector, 并基于之前定义的模式创建集合
24 collection_name = "word_vector"
25 collection = Collection(name=collection_name, schema=schema)
```



创建索引

创建索引时还可以设定以下参数用于设定索引方式：

- FLAT：准确率高，适合数据量小，暴力求解相似。
- IVF-FLAT：量化操作，准确率和速度的平衡
- IVF: inverted file 先对空间的点进行聚类，查询时先比较聚类中心距离，再找到最近的N个点。
- IVF-SQ8：量化操作，disk cpu GPU 友好
- SQ8：对向量做标量量化，浮点数表示转为int型表示，4字节->1字节。
- IVF-PQ：快速，但是准确率降低，把向量切分成m段，对每段进行聚类；查询时，查询向量分端后与聚类中心计算距离，各段相加后即为最终距离。使用对称距离(聚类中心之前的距离)不需要计算直接查表，但是误差会更大一些。
- HNSW：基于图的索引，高效搜索场景，构建多层的NSW。
- ANNOY：基于树的索引，高召回率

metric_type 包括 "IP" 内积、"COSINE" 余弦相似度、"L2" L2距离

```
1 # 创建索引
2 # 导入必要的类和模块
3 from pymilvus import Collection, utility, connections, db
```

```
4
5 # 连接到 Milvus 服务
6 conn = connections.connect(host="127.0.0.1", port=19530)
7
8 # 使用名为 sample_db 的数据库
9 db.using_database("sample_db")
10
11 # 定义索引参数，这里指定度量类型为余弦相似度 (COSINE)
12 index_params = {
13     "metric_type": "COSINE",
14 }
15
16 # 获取名为 "word_vector" 的集合
17 collection = collection("word_vector")
18
19 # 在名为 "embedding" 的字段上创建索引，索引参数为之前定义的
20 # index_params
21 collection.create_index(
22     field_name="embedding",
23     index_params=index_params
24 )
25
26 # 获取 "word_vector" 集合索引构建的进度
27 utility.index_building_progress("word_vector")
```

The screenshot shows the Milvus UI interface. On the left, there's a sidebar with icons for database management. The main area displays the 'sample_db' database, specifically the 'word_vector' collection. The collection has 0 entities. The 'overview' tab is selected, showing the following details:

- 描述 (Description): Test embedding search
- 状态 (Status): 未加载 (Not Loaded)
- 大约的Entity数量 (Approximate Entity Count): 0
- 创建时间 (Create Time): 2024/10/15 15:52:04
- 一致性 (Consistency): Bounded

Below this, under 'schema', is a table showing the collection's structure:

字段 (Field)	类型 (Type)	索引名称 (Index Name)	索引类型 (Index Type)	索引参数 (Index Parameters)	描述 (Description)
id	Int64		+ 创建索引 (Create Index)	--	
embedding	FloatVector(1024)	embedding	AUTOINDEX	metric_type: COSINE	

插入数据

将embedding后的数据存入向量数据库。

```
1 # 插入数据
2 # 导入必要的类和模块
3 from pymilvus import Collection, db, connections
4
5 # 连接到 Milvus 服务
6 conn = connections.connect(host="127.0.0.1", port=19530)
7
8 # 使用名为 sample_db 的数据库
9 db.using_database("sample_db")
10
11 # 指定要操作的集合名称
12 coll_name = 'word_vector'
13
14 # 获取名为 coll_name 的集合
15 collection = Collection(coll_name)
16
17 # 插入数据, 假设 embeddings 是一个已经准备好的向量列表或数组
18 mr = collection.insert([embeddings])
19
20 # 打印插入操作的结果
21 print(mr)
```

查询并检索

连接Milvus 服务将之前创建的集合加载到内存中，设定搜索参数执行搜索。

```
1 # 向量搜索
2 # 导入必要的类和模块
3 from pymilvus import Collection, db, connections
4 from pymilvus.orm import utility
5
6 # 连接到 Milvus 服务
7 conn = connections.connect(host="127.0.0.1", port=19530)
8
9 # 使用名为 sample_db 的数据库
10 db.using_database("sample_db")
11
```

```
12 # 指定要操作的集合名称
13 coll_name = 'word_vector'
14
15 # 获取名为 coll_name 的集合
16 collection = collection(coll_name)
17
18 # 将集合加载到内存中，以便进行搜索操作
19 collection.load()
20
21 # 定义搜索参数，这里指定度量类型为余弦相似度 (COSINE)
22 search_params = {"metric_type": "COSINE"}
23
24 # 假设 query_embedding 已经是一个包含查询向量的列表或数组
25 # 执行搜索操作
26 results = collection.search(
27     data=query_embedding.tolist(),           # 将查询向量转换为列表形式
28     anns_field="embedding",                 # 指定要搜索的向量字段名
29     param=search_params,                   # 搜索参数
30     limit=5,                            # 返回最多的结果数量
31     expr=None                           # 可选的过滤表达式，默认为 None
32     表示不过滤
33 )
34 # 输出搜索结果
35 for result in results:
36     print(result)
37
38 # ['id: 453238947178891095, distance: 0.6703383326530457,
39 # entity: {}',
40 # 'id: 453238947178891094, distance: 0.45103031396865845,
41 # entity: {}',
42 # 'id: 453238947178891096, distance: 0.08964216709136963,
43 # entity: {}']
```