（一）、简介

1.1 背景介绍

向量是一种将实体和应用代数化的表示，其将实体间的关系抽象成向量空间中的距离，而距离的远近代表着相似程度。向量检索便是对这类代数化的数据进行快速搜索和匹配的方法。

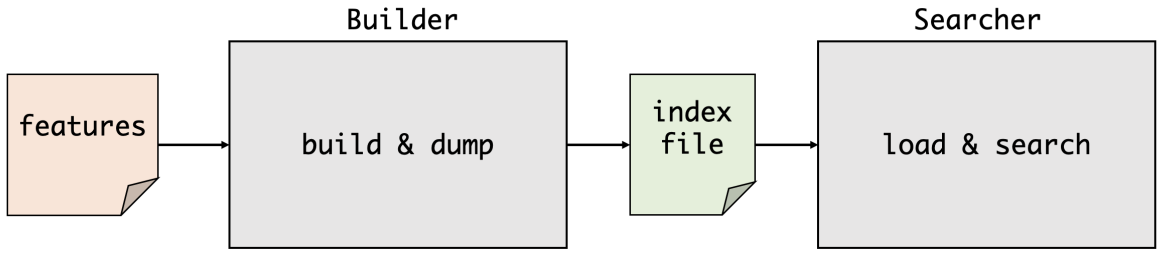
向量检索常涉及到的问题有 KNN 和 RNN，KNN（K-Nearest Neighbor）查找离查询点最近的 K 个点，而 RNN (Radius Nearest Neighbor) 查找查询点某半径范围内的所有点或 N 个点。

在涉及到大数据量的情况下，有些向量检索方法无法保证百分之百准确地取得 KNN 或 RNN，只能尽可能地或按概率的方式满足检索需求，因此大数据量检索实际要解决的是 ANN（Approximate Nearest Neighbor）的问题。

1.2 Proxima 介绍

Proxima 向量检索引擎是一个运用于大数据下，实现向量近邻搜索的高性能软件库。
该引擎采用插件式架构，为开发者提供了统一的软件接口和框架，并囊括了多种检索方法，如 BF（Brute Force）、PQ（Product Quantization）、HC（Hierarchical Clustering）、HNSW（Hierarchical Navigable Small World）、QGraph（Quantization Graph）、SSG（Satellite System Graph）等。同时，引擎对向量检索的一些基础能力，如聚类、距离计算、高并发、Cache等做了深层次的优化。

Proxima 向量检索的基本流程如下图所示：



首先从原始数据中读取特征向量，然后根据这些特征向量构建索引并 dump 到索引文件中；向量检索时加载该索引数据，并执行相应的向量检索操作。

本次比赛中，为参赛者提供了一个极简版的 Python SDK（Proxima 底层实现使用的是 C++ 语言），供参赛者使用。该版本支持插件化加载模块，当有新的插件功能时，可无须升级 Proxima，直接加载新的插件模块即可。参赛者也可以通过编写插件的方式编写更优的算法，加载自己编写的算法来提高向量检索的性能和召回率。

（二）、环境搭建

- Linux 操作系统下安装 Python3.7 （**注意：此次比赛仅提供 python3.7 版本，请务必确认**

python 版本）

- 下载 Proxima Python 安装包
- 使用 pip3 安装

Proxima 安装示例

```
# 使用 pip3 安装安装包
pip3 install --user pyproxima2-2.2.0-lite-cp37-cp37m-linux_x86_64.whl
```

（三）、使用示例

```
#encoding=utf8

"""
Example of how to use Proxima Python SDK
"""

from pyproxima2 import *
import numpy as np

class Example:
  def __init__(self):
    self.docs = 100000
    self.dim = 64
    self.topk = 10
  def prepare_data(self):
    """
    Insert data into holder
    """
    self.holder = IndexHolder(type=IndexMeta.FT_FP32, dimension=self.dim)
    for i in range(self.docs):
      vec = np.random.uniform(low=0.0, high=1.0, size=64).astype('f')
      self.holder.emplace(i, vec)

  def build(self):
    """
    Build index
    """
    builder = IndexBuilder(
        name="ClusteringBuilder",
        meta=IndexMeta(type=IndexMeta.FT_FP32, dimension=self.dim),
        params={'proxima.hc.builder.max_document_count': self.docs}
        )
    builder.train_and_build(self.holder)
    dumper = IndexDumper(path="./index/example.index")
    builder.dump(dumper)
    dumper.close()
```

```
  def search(self):
    """
    Do knn search
    """
    path = "./index/example.index"

    # init container
    container = IndexContainer(name='MMapFileContainer', params={})
    container.load(path)

    # init searcher
    searcher = IndexSearcher("ClusteringSearcher")
    ctx = searcher.load(container).create_context(topk=10)

    # do search
    vec = np.random.uniform(low=0.0, high=1.0, size=64).astype('f')
    results = ctx.search(query=vec)
    for ele in results[0]:
      print("Result: the key is: {0}\tscore is: {1}.".format(ele.key(),
ele.score()))

def main():
  exp = Example()
  exp.prepare_data()
  exp.build()
  exp.search()

if __name__=='__main__':
  main()
```

（四）、IndexHolder

以上示例中，我们通过 numpy 随机生成的 100000 个向量并插入到 IndexHolder 对象。在 Proxima 向量检索引擎中，我们把 IndexHolder 对象当作数据输入的适配层。首先需要读取数据到 IndexHolder 对象中，后续的 Train、Build 等模块都是以该数据对象为输入。

本小节简单介绍一些 IndexHolder 接口。通过该小节的学习，参赛者能够学会如何把数据输入到 Proxima 中。

4.1 初始化 IndexHolder 对象

```
IndexHolder(type=0, dimension=0, multipass=False, shallow=False)
```

参数名字	参数类型	描述
type	int	数据类型
dimension	int	向量维度
multipass	bool	holder是否使用可重复读取。一般一份holder多处使用，建议设置为True。如果只用一次，例如只train或只build，设置为False
shallow	bool	如果设置为True，则emplace的特征向量buffer不会copy一份，只保存引用，可节省内存，但要确保这些buffer使用中不会提前释放

目前支持的数据类型有：

数据类型名	数据类型（对应的 C++ 底层存储类型）
IndexMeta::FT_INT8	int8_t
IndexMeta::FT_INT16	int16_t
IndexMeta::FT_FP32	float
IndexMeta::FT_FP64	double
IndexMeta::FT_BINARY32	uint32_t
IndexMeta::FT_BINARY64	uint64_t

4.2 插入向量

```
emplace(key, data)
```

参数名字	参数类型	描述
key	long	向 IndexHolder 对象中插入一条向量
data	buffer obj	支持 Buffer protocol 的对象，例如：bytes、bytearray、array.array、numpy.array

4.3 获取 IndexHolder 中的数据特征

```
count()
```

描述	返回值	返回值类型
获取 IndexHolder 中向量个数	IndexHolder 中向量个数	int

```
dimension()
```

描述	返回值	返回值类型
获取 IndexHolder 中向量维度	IndexHolder 中向量维度	int

```
type()
```

描述	返回值	返回值类型
获取 IndexHolder 中数据类型	IndexHolder 中数据类型	int

4.4 示例

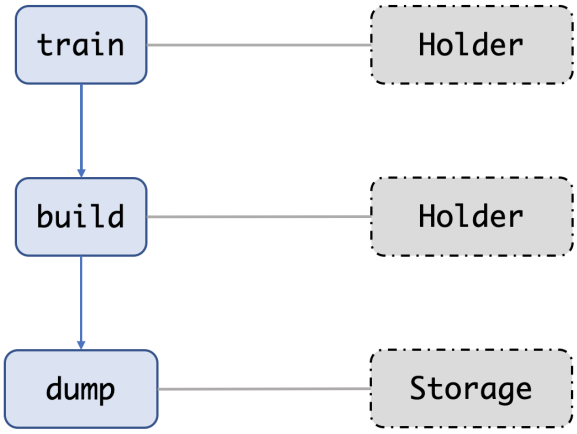
从文件中读取数据到 IndexHolder 对象中：

```
inFile = open("test.txt", "r")
holder = IndexHolder(type=IndexMeta.FT_FP32, dimension=512)
iNumLines = 0
while True:
    line_raw = inFile.readline()
    if(not line_raw):
        break

    line = line_raw.strip()
    if(line==""): continue

    fields = line.split(sep=",")
    key = int(fields[0])
    feature = np.array([np.float32(ele) for ele in fields[1].split(sep=" ")])
    holder.emplace(key, feature)
```

（五）、Build

Build 主要的职能是构建索引。其基本流程包括以下三点：

- 数据训练
- 构建索引
- dump 索引



在 Python SDK 中这三个过程可以在一行 Python 语句实现：

```
builder.train_and_build(self.holder).dump(IndexDumper(path="./index/example.index"))
```

以上语句会根据上一小节得到的 IndexHolder 对象进行数据训练、构建索引，并把构建的索引 dump 到 path 所指定的路径中，该索引文件可被后续的 Searcher 对象用于向量检索。

5.1 初始化 Builder 对象

```
builder = IndexBuilder(
    name="ClusteringBuilder",
    meta=IndexMeta(type=IndexMeta.FT_FP32, dimension=512),
    params={'proxima.hc.builder.max_document_count': 10000}
)
```

参数名字	参数类型	描述
name	str	构建索引使用的聚类器名称
meta	int	IndexMeta 对象，描述索引相关信息
params	dict	Builder参数

5.2 训练并构建索引

```
train_and_build(holder)
```

参数名字	参数类型	描述
holder	IndexHolder	持有特征数据的 IndexHolder 对象

该方法对 IndexHolder 对象中的数据进行训练并构建索引。

5.3 dump 索引到指定路径

可以通过 IndexDumper 对象输出索引到指定路径，IndexDumper 对象的初始化如下代码所示：

```
dumper = IndexDumper(path="./index/builder_example.index")
```

其中，path 指定需要保存的索引路径。得到 IndexDumper 对象后，可以调用 IndexBuilder 中的 dump 接口来保存训练好的索引：

```
builder.dump(dumper)
```

注意：使用 IndexDumper 对象 dump 索引后需要对 IndexDumper 对象执行 close 操作：

```
dumper.close()
```

5.4 示例

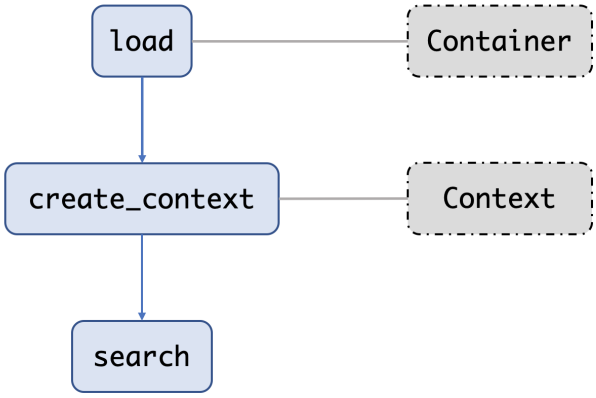
```
builder.train_and_build(holder)
dumper = IndexDumper(path="./index/builder_example.index")
builder.dump(dumper)
dumper.close()
```

（六）、Searcher

IndexSearcher 是进行 KNN 检索的主要模块，以只读方式加载离线构建好的索引并进行在线检索。IndexSearcher 使用流程：

- 加载索引数据
- 创建检索上下文
- 执行查询

如下图所示：



6.1 初始化 IndexSearcher 对象

```
searcher = IndexSearcher(name='', params={})
```

参数名字	参数类型	描述
name	str	IndexSearcher 名称：目前支持（本次比赛所提供）：ClusteringSearcher 和 HnswSearcher
params	dict	Builder 参数

6.2 加载索引数据

Proxima 通过 IndexContainer 对象来加载索引数据。IndexContainer 是用来加载索引的容器，它屏蔽了底层存储系统差异，以及 IndexSearcher 与存储系统交互的差异。

IndexContainer 的初始化：

```
container = IndexContainer(name="MMapFileContainer", path="", params={})
```

参数名字	参数类型	描述
name	str	索引容器名称
path	str	索引路径
params	dict	每种Container支持的参数

其中索引容器名称可选的有多种，为了方便接口，目前提供给参赛者的是 MMapFileContainer：

- MMapFileContainer 通过 mmap 方式加载索引。

在准备好 IndexContainer 对象后， IndexSearcher 对象可以加载该 IndexContainer 对象：

```
searcher.load(container)
```

6.3 创建检索上下文

IndexContext 用于保存检索上下文内容，包含检索参数，内部状态，检索结果等内容。在进行检索之前需要首先创建检索上下文。

```
create_context(topk=None, debug=False, filter=None)
```

参数名字	参数类型	描述
topk	int	向量检索需要召回的向量个数
debug	bool	是否开启debugMode （参赛者可忽略）
filter	callable obj	过滤器，可设置为 lambda 函数。参数为 long 类型的 key。如须过滤返回 True，例如：ctx.set_filter(lambda key: key < 5) （参赛者可忽略）

6.4 执行向量检索

在以上准备工作都完成后，用户可以通过 IndexContext 对象执行向量检索操作：

```
search(query)
```

参数名字	参数类型	描述
query	object	支持Buffer protocol的对象，例如：bytes、bytearray、array.array、numpy.array
count	int	如果 count > 1，则表示进行批量查询，query 为多个连续待检索的 query
type	int	数据类型，见 4.1 节所述
dimension	int	query 维度，默认为索引特征维度

6.5 示例

```
# init searcher
searcher = IndexSearcher("ClusteringSearcher")

# init container
container = IndexContainer(name='MMapFileContainer',
                           path="./index/builder_example.index",
                           params={})

# load index
searcher.load(container)

# create search context
ctx = searcher.create_context(topk=10)

# do search
vec = np.random.uniform(low=0.0, high=1.0, size=512).astype('f')
results = ctx.search(query=vec)
for ele in results[0]:
    print("Result: the key is: {0}\tscore is: {1}.".format(ele.key(),
ele.score()))
```

注意：在本示例中，最后打印结果时使用的是 `results[0]` 而不是 `results`。因为本例中使用的是单个 query 进行检索，结果保存到 `results[0]` 中，当有多个向量进行批量检索时，相应的结果依次保存到相应的 `results[i]` 位置中。

（七）、Plugin

Proxima 引擎采用插件式架构，为开发者提供了统一的软件接口和框架，针对不同场景，支持多种检索算法。当需要添加新的算法时，开发者只需要实现相应的插件，并在程序中加载该插件即可。参赛者可以通过编写插件的方式提供自己的检索算法。

本小节将简单介绍如何编写自己的插件，参赛者可以选择是否编写一个性能更好、召回率更高的算法插件。

注：本节是提供给参赛者的附加题，参赛者可以选做

7.1 示例

示例代码位置：plugin 文件夹

在代码的 src 和 include 中参赛者可以实现自己的 Plugin，目前也为参赛者提供了两个编写 Plugin 的示例，分别是 plugin_build.cc 和 plugin_search.cc。示例仅仅作为如何编写 Plugin 的参照，并不是已经完成的部分代码，参赛者完全可以编写自己的插件。

7.2 编写简述

在实现插件时，参赛者需要继承已有的基类。比如实现 Build 时需要继承 aitheta2::IndexBuilder，并实现其中的函数：

```
class ExampleBuilder : public aitheta2::IndexBuilder
{
  ...
}
```

在实现完相应的算法类后，参赛者需要注册相应的类，给参赛者提供的接口如 INDEX_FACTORY_REGISTER_BUILDER、INDEX_FACTORY_REGISTER_SEARCHER 等：

```
INDEX_FACTORY_REGISTER_BUILDER(ExampleBuilder);
INDEX_FACTORY_REGISTER_SEARCHER(ExampleSearcher);
```

最后参赛者需有修改 CMakeLists.txt 文件来生成库文件，例如：

```
add_library(hello_library SHARED
    src/plugin_build.cc
    src/plugin_searcher.cc
)
```

当参赛者编写完插件后需要在自己的程序中通过加载插件的方式加载自己的插件。例如：

```
IndexPlugin.load("<path to you .so file>/lib<you library name>.so")
```