探索式数据分析大作业实验报告

数据科学导论 2023-2024 春季学期

1. 引言

随着大数据时代的到来，我们积累了大量的数据，这些数据包含了有价值的信息，但也常常隐藏在海量的数据中。探索式数据分析可以帮助我们有效地理解和利用这些数据，发现其中的潜在模式、趋势和关联，从而为决策提供支持。

面对大规模数据时，精确查询可能会变得非常耗时。近似查询处理（AQP）问题不要求完全精确的匹配，要求在保证一定结果精度的情况下，通过采样、直方图、深度学习等方法，快速地统计给定限制内数据的特征（如计数、平均值、求和），以提高数据分析的速度。

笔者设计了利用 KD-Tree 维护直方图的快速近似查询处理方案。在设计过程中，方案面临着性能、精度、资源占用的多重挑战，以 Python 语言限制带来的性能挑战为主。
大体上来说，利用库函数和多线程+多进程并行计算来优化 Python 较差的性能，以牺牲内存和时间来提升近似查询的精度，通过离线处理模型和线程/进程池来平衡时间和内存占用的限制，总代码量 18KB。

经过测试，本方案可以在给定的四个查询负载中，分别依次取得 MSLE $1.47\mathrm{e}-5$，$3.21e-6$，$1.58e-5$，$2.68e-5$ 的较高精度，同时单次 online 时间最高为 $2.679\,s$，单次 offline 时间最高为 $98\,s$。

2. 整体方案

KD-Tree 是一种可以高效处理 k 维空间信息的数据结构，可以在 $O\left(n^{1-1/k}\right)$ 的时间复杂度下查询 k 维空间的信息，在结点数 n 远大于 2^k 时运行效率较好。

分析给定的 Workload 信息可得，所有查询的约束空间不超过 3 个（不是若干个），且对于离散型维度的限制只有 1 个值，所以将 12 维全部拿来建树是十分浪费的。

我们的方案是，对于所有离散型维度的 $C_5^0+C_5^1+C_5^2+C_5^3$ 种限制情况分别建 KD-Tree，每棵树对 7 个连续型维度分割，这样降低了 KD-Tree 的维数，也进一步提升了精度。

实际测试发现 Python 建树的效率并不高，离精确查询相差甚远，于是每一棵 KD-Tree 都限制了树高，达到限制高度后不再继续分割，对叶节点的区间信息作近似查询。

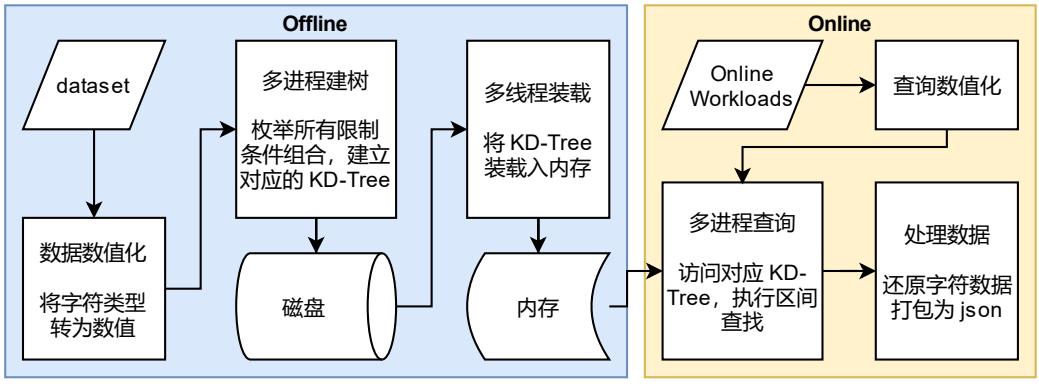

图 1　整体框架

图 1 展示了本方案的整体框架，其中 offline 只需要执行一次，用于对数据的预处理、建树和装载；online 会在每次询问时调用，用于处理询问得到答案。

3. 详细设计

3.1. 符号定义

符号	含义
$data$	数值化后的数据集
$Node()$	KD-Tree 节点类，内有 lc, rc, cnt, sum, dim，表示左右子节点、区间结点数、区间和以及区间边界
D	离散型数据维度数
$bound$	查询的区间范围
pre	数值化后的 $predicate$
$result$	返回的 $Answer$ 类答案，内部包含 $groupby$ 的 $groupAnswer$
d_{max}	KD-Tree 建树的最大限制深度

3.2. Offline 阶段

在评测开始时，依次调用 3 个函数：loadData，buildKDTrees，loadTreeData，完成预处理阶段总流程，流程图见图 2。

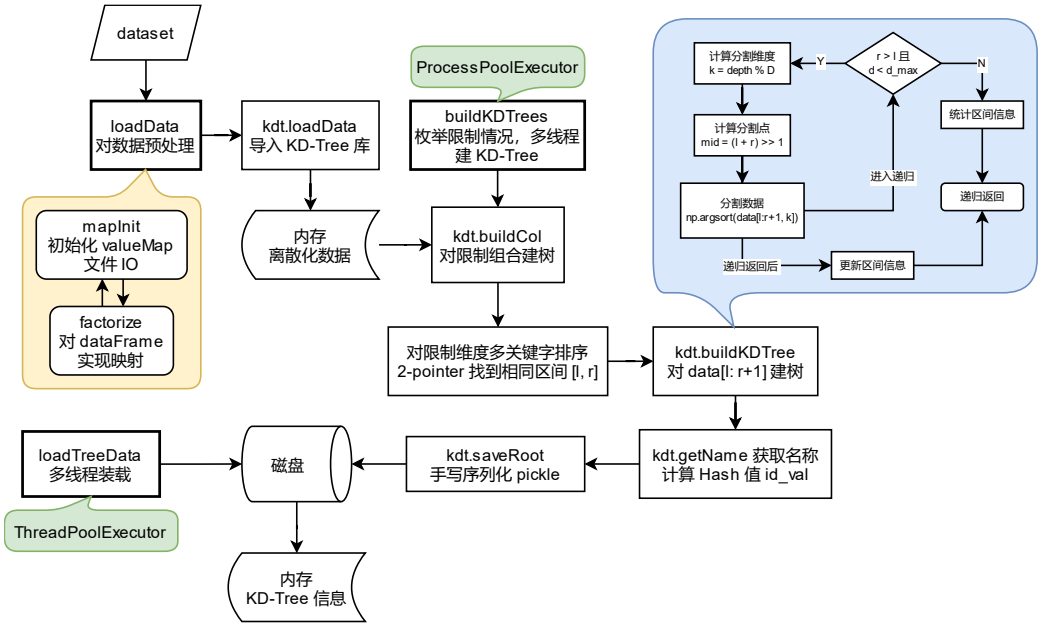

图 2 Offline 流程图

3.2.1. 数据集处理方法

在 kdt.buildCol 中，已知离散型限制 col，需要对满足 col 的所有取值情况分别建树。一种朴素的想法是枚举所有情况，然后在数据集里查找，但查找的复杂度较高，实际实现难以接受。

考虑面向数据集的扫描法，按照限制 col 对原数据集进行多关键字排序，这样相邻的两个数据在数据集中出现的情况相同或相近。采用 2-pointer 对排序后的数据进行扫描，便可在 $O(n \log n)$ 的时间复杂度内完成数据集的分割。

3.2.2. KD-Tree 递归建树

C 语言编写 KD-Tree 时会用到库函数 std::nth_element，而在 Python 中并没有相同的库函数。一种性能较优的实现是使用 numpy 库的 np.argsort，建树复杂度变为 $O(n \log^2 n)$，若直接使用 list 的 sort 加上 lambda，其运行用时约为 np.argsort 的两倍以上。由于 Python 的 List 内部元素地址并不连续，在涉及大规模数据时，使用 np.ndarray 便会有较大的效率提升。

Algorithm 1 buildKDTree

Require: $d_{\max}$

function BUILD($data, l, r, depth$)
  $u \leftarrow$ new **Node()**
  if $l == r$ **or** $depth \geq d_{\max}$ **then**
    for $i \leftarrow [l, r]$ **do**
      $u.cnt \mathrel{+}= 1$
      $u.dim = u.dim \cup data[i]$
      for $j \leftarrow [0, D)$ **do**
        $u.sum \mathrel{+}= data[i][j]$
      end for
    end for
    return u
  end if
  $k \leftarrow depth\%D$
  $mid \leftarrow \frac{l+r}{2}$
  SPLIT($data, l, r, mid$)
  $u.cnt \leftarrow r - l + 1$
  $u.lc \leftarrow$BUILD($data, l, mid, depth + 1$)
  $u.rc \leftarrow$BUILD($data, mid + 1, r, depth + 1$)
  PUSHUP($u, u.lc$)
  PUSHUP($u, u.rc$)
  return u
end function

3.2.3. 序列化装载

由于 Q1~Q4 用的是同一个 dataset，一个自然的想法是用一次 offline 建树后不再建树，于是可以将建好的 KD-Tree 装载入本地，再次调用 offline 阶段时只 loadTreeData。由于 Python 的文件 IO 效率并不高，采用多进程来优化建树并装载过程，但多进程的数据会随着进程结束而被回收，无法共享到主进程，于是建树完成之后的进程也需要本地装载。

Python 的 json 库和 pickle 库提供了将 class 类型序列化装载入文件的库函数，但由于要写入的 KD-Tree 森林总内存约 1GB，库函数的实现效率过低（写入 200MB 用时约 30s），需要手动实现序列化装载。将 class Node 中的所有元素转换为 list，用 file.write 装载，用时仅为 pickle 的 30%。

3.3. Online 阶段

图 3 展示了 Online 阶段的流程图。

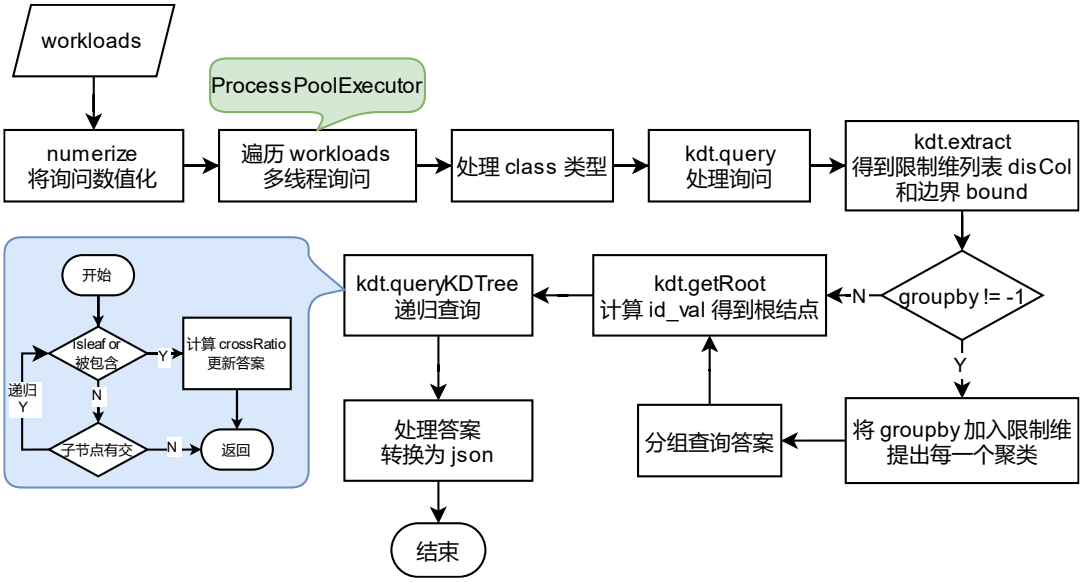

图 3 Online 流程图

3.3.1. query 实现

在实际测试中，发现 YEAR_DATE 一项的误差值较大，原因是 YEAR_DATE 的值均为整数，而询问的 *bound* 边界是随机浮点数，在用区间长度来估算 *crossRatio* 时会有较大的误差。

对限制为 YEAR_DATE 的 $predicate$ 边界取整处理，最后计算的时候补上整数对应区间长度的 1，修改后 4 组 workload 的平均 MSLE 下降了约 40%。

Algorithm 2 query for count*

```
function QUERY(pre, groupby)
    if groupby ≠ −1 then
        for i ← (0, num_groupby) do
            pre_i ← pre ∧ (groupby = i)
            result_i ← QUERY(pre_i, −1)
        end for
        return result
    else
        col ← pre_discrete
        bound ← EXTRACT(pre_continous)
        id_val ← GETHASH(col)
        root ← GETROOT(id_val)
        return QUERYKDTREE(root, bound)
    end if
end function
function QUERYKDTREE(u, bound)
    if u = ∅ or u.dim ∩ bound = ∅ then
        return 0
    end if
    if u is Leaf or u.dim ⊆ bound then
        ratio = ∀(u.dim∩bound) / ∀(u.dim)
        return u.cnt × ratio
    end if
    return QUERYKDTREE(u.lc, bound) + QUERYKDTREE(u.rc, bound)
end function
```

4. 实验结果及分析

4.1. 整体实验结果

整体实验结果见表 1，平均 MSLE 为 1.51e − 5，实验环境如下：

- 系统：Ubuntu 20.04.3 LTS (GNU/Linux 5.15.79.1-WSL2 x86_64)。
- CPU：12th Gen Intel(R) Core(TM) i9-12900H 2.50 GHz。
- 内存：16GB DDR5 4800。
- 硬盘：1TB NVMe PCIe 4.0 SSD。

表 1 整体实验结果

查询负载	误差（MSLE）	在线时间	离线时间
Q1	1.47e − 5	0.778 s	98.05 s
Q2	3.21e − 6	0.989 s	−
Q3	1.58e − 5	1.526 s	−
Q4	2.68e − 5	2.679 s	−

实验时受内存限制，多进程询问 max_workers=4，其余为 8，内存峰值约 9.6 GB。若测试时内存超限，调用虚拟内存效率低，Q4 的在线时间会有较大波动（见 4.2.2.）。

4.2. 参数分析

4.2.1. 树高调整

Offline 阶段的 buildKDTrees 中，传入了参数 $deltaDepth$，表示与实际最高 KD-Tree 树高的差值，用此参数来限制 KD-Tree 森林的树高，并保证相对精度。

我们采用 Q4 查询负载的 MSLE、online_time 和 offline_time 来评估 deltaDepth 的作用效果，图表见图 4。可以发现 $deltaDepth$ 每 +1，代表 KD-Tree 森林的树高相对全部 +1，体现在 MSLE 上的效果是 × 50%。

最终的提交测试中，权衡误差-时间取舍，取 $deltaDepth = -3$。

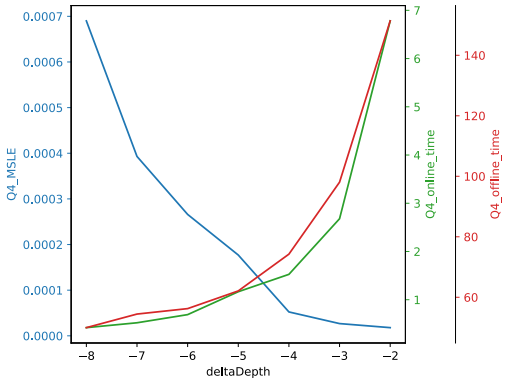

图 4 *deltaDepth* 参数对应的 Q4 表现

4.2.2. 进程数调整

在 online 的询问中，我们采用了多进程并行计算来加速询问，进程数（max_workers）与 online_time 的对应关系见图 5。

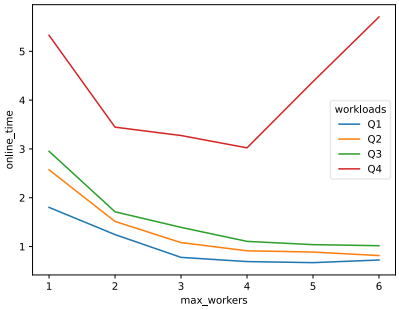

图 5 进程数对应的 online 表现

可以发现在 max_workers 取 5 及以上时，Q4_online_time 有较大幅度的增长，原因是评测环境内存受限，调用虚拟内存影响运行速度。

考虑到最终评测环境为 80 核 1TB 内存，最终提交 max_workers 取 8。

5. 总结及体会

本实验实现最难的地方便是如何在 Python 环境下优化运行速度，几处关键优化在代码中以注释的形式保留体现。使用 Python 编写本实验项目也颇具挑战，对于数据结构的设计需要结合自身语言特点重新考虑。

附录：

代码仓库：https://www.github.io/sheriyuo/AQP-KDTree

代码文件：aqp.py
import pandas as pd import json import os import sys from os import path as osp from tqdm import tqdm from concurrent.futures import ProcessPoolExecutor lib_dir = osp.join(osp.dirname(osp.abspath(__file__)), 'codes') sys.path.append(lib_dir)

```
import kdtlib as lib

def aqp_online(data: pd.DataFrame, Q: list) -> list:
    Q = pd.json_normalize([json.loads(i) for i in Q]) # 把 Q 转为 DataFrame
    Q = lib.numerize(Q) # 数值化 Q

    results = []

    with ProcessPoolExecutor(max_workers=4) as executor:
        futures = []
        for _, q in Q.iterrows():
            futures.append(executor.submit(lib.query, q))

        for future in tqdm(futures):
            results.append(future.result())

    results = [json.dumps(i, ensure_ascii=False) for i in results]

    return results

def aqp_offline(data: pd.DataFrame, Q: list) -> None:
    lib.loadData(data) # 读入数据
    lib.buildKDTrees(-3) # 预处理建树
    lib.loadTreeData() # 读入 KDTree 数据
```

代码文件：codes/kdtlib.py

```
import numpy as np
import pandas as pd
import os
import os.path as osp
import sys
from itertools import combinations, product
from concurrent.futures import ProcessPoolExecutor
from tqdm import tqdm
from scipy.special import comb
import shutil

LIB_DIR = osp.dirname(osp.abspath(__file__))
WORKING_DIR = osp.dirname(osp.dirname(osp.abspath(__file__)))
DATA_DIR = osp.join(WORKING_DIR, "data")
if not osp.exists(DATA_DIR):
    os.mkdir(DATA_DIR)

sys.path.append(LIB_DIR)
```

```
import kdt

COLUMNS = [
    "YEAR_DATE",
    "DEP_DELAY",
    "TAXI_OUT",
    "TAXI_IN",
    "ARR_DELAY",
    "AIR_TIME",
    "DISTANCE",
    "UNIQUE_CARRIER",
    "ORIGIN",
    "ORIGIN_STATE_ABR",
    "DEST",
    "DEST_STATE_ABR",
]
DISCRETE_COLUMNS = [
    "UNIQUE_CARRIER",
    "ORIGIN",
    "ORIGIN_STATE_ABR",
    "DEST",
    "DEST_STATE_ABR",
]
CONTINUOUS_COLUMNS = [
    "YEAR_DATE",
    "DEP_DELAY",
    "TAXI_OUT",
    "TAXI_IN",
    "ARR_DELAY",
    "AIR_TIME",
    "DISTANCE",
]
OP_MAP = {"count": 0, "sum": 1, "avg": 2}

COLUMN2INDEX = {c: i for i, c in enumerate(COLUMNS)}
COLUMN2INDEX.update({"_None_": -1})
COLUMN2INDEX.update({"*": -1})

DATA, ID2VALUE, VALUE2ID = None, None, None

modelNames = []

# 对 dataFrame 实现映射
def factorize(df, columns=DISCRETE_COLUMNS):
```

```
    id2value, value2id = {}, {}
    df = df.copy()
    for col in columns:
        value, index = pd.factorize(df[col])
        df[col] = value
        col_id = COLUMN2INDEX[col]
        id2value.update({col_id: index.values})
        value2id.update({col_id: {v: i for i, v in enumerate(index)}})
        id2value.update({col: index.values})
        value2id.update({col: {v: i for i, v in enumerate(index)}})

    return df, id2value, value2id

# 初始化 valueMap
def mapInit(data=None):
    global DATA, ID2VALUE, VALUE2ID
    # 如果已经本地装载
    if osp.exists(osp.join(DATA_DIR, "maps.npy")) and osp.exists(
            osp.join(DATA_DIR, "dataset.csv")
        ):
        DATA = pd.read_csv(osp.join(DATA_DIR, "dataset.csv"))
        VALUE2ID, ID2VALUE = np.load(
            osp.join(DATA_DIR, "maps.npy"), allow_pickle=True
        )
        return
    if data is None:
        return
    # 装载到本地
    DATA, ID2VALUE, VALUE2ID = factorize(data)
    maps = [VALUE2ID, ID2VALUE]
    np.save(osp.join(DATA_DIR, "maps.npy"), maps)
    DATA.to_csv(osp.join(DATA_DIR, "dataset.csv"), index=False)

# 根据 col 读取模型名称
def getDataName(col) -> str:
    name = ""
    for c in col:
        name += str(c)
    return name

# 将询问数值化
def numerize(Q: pd.DataFrame):
    # result_col
    _result = []
```

```
    for col in Q["result_col"]:
        _result.append(
            [(OP_MAP[i[0]], COLUMN2INDEX[i[1]]) for i in col if len(i) >= 2]
        )
    Q["result_col"] = _result

    # predicate
    _predicate = []
    for predicate in Q["predicate"]:
        cur_predicate = []
        for i in predicate:
            col = COLUMN2INDEX[i["col"]]
            lb, ub = 0, 1e9
            if i["col"] in DISCRETE_COLUMNS:
                lb = ub = VALUE2ID[i["col"]][i["lb"]]
            else:
                if i["lb"] != "_None_":
                    lb = float(i["lb"])
                if i["ub"] != "_None_":
                    ub = float(i["ub"])
            cur_predicate.append((col, lb, ub))
        _predicate.append(cur_predicate)
    Q["predicate"] = _predicate

    # groupby
    _groupby = Q["groupby"].apply(COLUMN2INDEX.get)
    Q["groupby"] = _groupby

    return Q

# 从 data 中读取数据
def loadData(data):
    if not osp.exists(osp.join(DATA_DIR, "dataset.csv")):
        data = data[COLUMNS]
        mapInit(data)
        data = DATA.copy()
        values = data.values.astype(np.float32)
        kdt.loadData(values, data.shape[0])

def buildKDTreeProcessed(col, deltaDepth):
    _col = np.array(col, dtype=np.int32)
    kdt.buildCol(_col, len(_col), deltaDepth)

def buildKDTrees(deltaDepth=0):
```

```
    global modelNames
    if osp.exists(osp.join(DATA_DIR, "list.txt")):
        return

    # 多进程数据本地装载
    with ProcessPoolExecutor(max_workers=8) as executor:
        futures = []
        for num in [0, 1, 2, 3]:
            for col in combinations(range(7, 12), num):
                futures.append(executor.submit(
                    buildKDTreeProcessed, col, deltaDepth
                ))
                modelNames.append(getDataName(col))

        for future in tqdm(futures):
            future.result()

    with open(osp.join(DATA_DIR, "list.txt"), "w") as file:
        for name in modelNames:
            file.write(name + "\n")

def query(Q):
    res = np.array(
        [kdt.Result(op, col) for op, col in Q["result_col"]]
    )
    pre = np.array(
        [kdt.Predicate(col, lb, ub) for col, lb, ub in Q["predicate"]]
        )

    groupby = Q['groupby']
    ans = kdt.query(res, len(res), pre, len(pre), groupby)
    size = ans.size
    ret = []
    # 处理 groupby
    for i in range(size):
        g_ans = ans.group_ans[i]
        if g_ans.id < 0:
            ret.append([g_ans.value])
        else:
            id = g_ans.id
            ret.append([ID2VALUE[groupby][id], g_ans.value])
    return ret

isLoaded = 0
```

```
def loadTreeData():
    global isLoaded
    if isLoaded == 0:
        kdt.loadTreeData()
        isLoaded = 1

mapInit()
```

代码文件：codes/kdt.py

```
import numpy as np
from enum import Enum
import math
import os.path as osp
from tqdm import tqdm
from concurrent.futures import ThreadPoolExecutor
LIB_DIR = osp.dirname(osp.abspath(__file__))
WORKING_DIR = osp.dirname(osp.dirname(osp.abspath(__file__)))
DATA_DIR = osp.join(WORKING_DIR, "data")

M = 12
D = 7
MOD = 1000000007
COL_NUM = [1, 1, 1, 1, 1, 1, 1, 26, 363, 53, 366, 53]

class OP(Enum):
    COUNT = 0
    SUM = 1
    AVG = 2

INT_T = int
OP_T = OP
FLOAT_T = float

# KD-Tree 节点
class Node:
    def __init__(self):
        self.lc = None
        self.rc = None
        self.cnt = 0
        self.sum = [0] * D
        self.dim = [[1e9, -1e9] for _ in range(D)]

# AQP 询问参数
```

```
class Result:
    def __init__(self, op: OP_T, col: INT_T):
        self.op = op
        self.col = col

class Predicate:
    def __init__(self, col: INT_T, lb: FLOAT_T, ub: FLOAT_T):
        self.col = col
        self.lb = lb
        self.ub = ub

# AQP 询问答案
class GroupAnswer:
    def __init__(self, id=0, value=0.0):
        self.id = id
        self.value = value

class Answer:
    def __init__(self, group_ans=None, size=0):
        self.group_ans = group_ans if group_ans is not None else []
        self.size = size

rootMap = {}
disCol = []
rootName = []

pdata = None
data = None
maxDepth = None

n = 0
sum = [0.0] * D
cnt = 0.0

def isLeaf(u: Node):
    return u.lc is None and u.rc is None

def isContinuous(c):
    return c <= 6

def isDiscrete(c):
    return c >= 7

def getId(i, j):
```

```
    return i * M + j

# 将 _data 导入为一维 pdata
def loadData(_data, _n):
    global n, pdata, data
    n = _n
    pdata = _data.reshape(-1)[:n * M]

# 更新 KD-Tree 节点区间信息
def updateNode(u, v):
    if v is None:
        return

    for i in range(len(u.sum)):
        u.sum[i] += v.sum[i]
        u.dim[i][0] = min(u.dim[i][0], v.dim[i][0])
        u.dim[i][1] = max(u.dim[i][1], v.dim[i][1])

# 取 data[l: r+1] 建立 KD-Tree
def buildKDTree(data, l, r, depth):
    if l > r:
        return None

    u = Node()
    if l == r or depth >= maxDepth:
        u.lc = None
        u.rc = None
        u.cnt = r - l + 1
        u.sum = [0] * D
        u.dim = [[1e9, -1e9] for _ in range(D)]
        for i in range(D):
            u.sum[i] += np.sum(data[l:r + 1, i])
            u.dim[i][0] = min(u.dim[i][0], np.min(data[l:r + 1, i]))
            u.dim[i][1] = max(u.dim[i][1], np.max(data[l:r + 1, i]))

            # 优化前的实现：
            # for j in range(l, r + 1):
            #     u.sum[i] += data[j][i]
            #     u.dim[i][0] = min(u.dim[i][0], data[j][i])
            #     u.dim[i][1] = max(u.dim[i][1], data[j][i])

        return u

    k = depth % D
```

```
    mid = (l + r) >> 1

    arg = np.argsort(data[l:r+1, k]) # 代替 nth_element 的较优实现
    data[l:r+1] = data[l:r+1][arg]

    # 优化前的实现：
    # data[l:r + 1] = sorted(data[l:r + 1], key=lambda x: x[k])

    u.cnt = r - l + 1
    u.lc = buildKDTree(data, l, mid, depth + 1)
    u.rc = buildKDTree(data, mid + 1, r, depth + 1)
    u.sum = [0] * D
    u.dim = [[1e9, -1e9] for _ in range(D)]
    updateNode(u, u.lc)
    updateNode(u, u.rc)

    return u

# 对 columns 的组合建树
def buildCol(col, size, delta_depth):
    global maxDepth, saveCount, rootName

    # 多关键字排序
    d = np.arange(n)
    if len(col) > 0:
        sort_indices = np.lexsort([pdata[getId(d, i)] for i in col])
        d = d[sort_indices]

    # 优化前的实现：
    # d = list(range(n))
    # d.sort(key=lambda x: [pdata[getId(x, col[i])] for i in range(size)])

    _data = np.zeros((n, D), dtype=np.float64)
    for i in range(n):
        _data[i] = pdata[d[i] * M:d[i] * M + D]

    name = getName(col)
    saveCount = 0
    with open(getPath(name), "w") as file:
        l = 0
        while l < n:
            id_val, base = 0, 1

            # 获取对应取值所建树的根节点 Hash 值
```

```
                j = 7
                for i in col:
                    while j < i:
                        base *= 13131
                        base %= MOD
                        j = j + 1
                    id_val += base * (pdata[getId(d[l], i)] + 1)
                    id_val %= MOD

                r = l   # 得到相同取值的区间 [l, r]
                while r < n - 1:
                    mismatch = False
                    for i in col:
                        if pdata[getId(d[l], i)] != pdata[getId(d[r + 1], i)]:
                            mismatch = True
                            break
                    if mismatch:
                        break
                    r += 1

                if size == 0: # 降低第一棵树的深度以减少 online 时间
                    delta_depth = delta_depth - 2
                maxDepth = max(1, int(math.log2(r - l + 1) + delta_depth))

                if size == 3: # 限制均为离散型时不需要建树
                    maxDepth = 0

                root = buildKDTree(_data, l, r, 0)
                id_val = int(id_val)
                rootMap[id_val] = root

                # 装载到本地
                saveRoot(root, id_val, file)

                l = r + 1

def getName(col) -> str :
    name = ""
    for i in col:
        name += str(i)
    return name

def getPath(name) -> str :
    return osp.join(DATA_DIR, "_" + name + ".txt")
```

```
def saveKDTree(u, file):
    if not file or not u:
        return

    # 手动实现 class Node 的序列化
    if u.lc is None:
        file.write(str(None))
    else:
        file.write("1")
    file.write(" ")
    if u.rc is None:
        file.write(str(None))
    else:
        file.write("1")
    file.write(" ")
    file.write(str(u.cnt) + " ")
    for x in u.sum:
        file.write(str(x) + " ")
    for x in u.dim:
        file.write(str(x[0]) + " " + str(x[1]) + " ")
    file.write("\n")

    # 优化前的实现：
    # pickle.dump(u, file)

    saveKDTree(u.lc, file)
    saveKDTree(u.rc, file)

def saveRoot(u, id, file):
    if not file or not u:
        return

    file.write(str(id) + "\n")
    saveKDTree(u, file)

def loadKDTree(_data, i):
    nodeData = _data[i].split()

    # 手动实现 class Node 的反序列化
    u = Node()
    u.cnt = int(nodeData[2])
    u.sum = [float(i) for i in nodeData[3: 10]]
    u.dim = [[float(nodeData[i]), float(nodeData[i+1])] for i in range(10, 24, 2)]
```

```
    # 优化前的实现：
    # pickle.load(u, file)

    if nodeData[0] == '1':
        u.lc, i = loadKDTree(_data, i + 1)
    if nodeData[1] == '1':
        u.rc, i = loadKDTree(_data, i + 1)
    return u, i

# 读取 name 的 KD-Tree 森林
def loadname(name):
    with open(osp.join(getPath(name)), "r") as file:
        _data = [line.strip() for line in file.readlines()]
    i = 0
    while i < len(_data):
        idx = int(_data[i])
        rootMap[int(idx)], i = loadKDTree(_data, i + 1)
        i = i + 1
    return

def loadTreeData():
    global rootName
    with open(osp.join(DATA_DIR, "list.txt")) as file:
        rootName = [name.strip() for name in file.readlines()]
    rootName = rootName[::-1]

    # IO 相关使用多线程加速，多进程无法保存
    with ThreadPoolExecutor(max_workers=8) as executor:
        futures = []
        for name in rootName:
            futures.append(executor.submit(loadname, name))

        for future in tqdm(futures):
            future.result()

# 处理得到 col 和 disCol
def extract(pre, psize, bound, col):
    global disCol
    disCol.clear()
    for i in range(psize):
        if isContinuous(pre[i].col):
            disCol.append(pre[i].col)
            if pre[i].col > 0:
```

```
                    bound[pre[i].col][0] = pre[i].lb
                    bound[pre[i].col][1] = pre[i].ub
                else: # YEAR_DATE 只会取整数值
                    bound[pre[i].col][0] = math.ceil(pre[i].lb)
                    bound[pre[i].col][1] = math.floor(pre[i].ub)
            else:
                col.append((pre[i].col, int(pre[i].lb)))

# 根据 col 求 Hash 值，得到根节点指针
def getRoot(col):
    col.sort()
    id_val, base = 0, 1
    i = 7
    for c, val in col:
        while i < c:
            base *= 13131
            base %= MOD
            i = i + 1
        id_val += base * (val + 1)
        id_val %= MOD

    if id_val not in rootMap:
        return None
    return rootMap[id_val]

# 近似询问区间的相交率
def crossRatio(dim, bound):
    ratio = 1.0
    for i in disCol:
        if dim[i][0] == dim[i][1]:
            ratio *= bound[i][0] <= dim[i][0] <= bound[i][1]
        else:
            l = max(dim[i][0], bound[i][0])
            r = min(dim[i][1], bound[i][1])
            t = 1 if i == 0 else 0 # YEAR_DATE
            ratio *= (r - l + t) / (dim[i][1] - dim[i][0] + t)
    return ratio

def isContain(dim, bound):
    for i in disCol:
        if dim[i][0] < bound[i][0] or dim[i][1] > bound[i][1]:
            return False
    return True
```

```
def isCross(dim, bound):
    for i in disCol:
        if dim[i][0] > bound[i][1] or dim[i][1] < bound[i][0]:
            return False
    return True

# KD-Tree 上递归查询
def queryKDTree(u, bound):
    global sum, cnt
    if u is None:
        return
    if isLeaf(u) or isContain(u.dim, bound):
        ratio = crossRatio(u.dim, bound)
        cnt += u.cnt * ratio
        for i in range(D):
            sum[i] += u.sum[i] * ratio
        return

    if u.lc and isCross(u.lc.dim, bound):
        queryKDTree(u.lc, bound)
    if u.rc and isCross(u.rc.dim, bound):
        queryKDTree(u.rc, bound)

def queryRange(root, bound):
    global sum, cnt
    sum.clear()
    sum.extend([0.0] * D)
    cnt = 0.0
    queryKDTree(root, bound)

# AQP query
def query(res, rsize, pre, psize, groupby):
    global sum, cnt
    bound = [[0.0, 0.0] for _ in range(D)]
    col = []
    ans = Answer()

    extract(pre, psize, bound, col)

    if groupby != -1: # 存在 groupby，加入限制内
        index = -1
        in_pre = any(p[0] == groupby for p in col)
        if not in_pre:
            ans.size = COL_NUM[groupby] * rsize
```

```
            col.append((groupby, -1))
        else:
            ans.size = rsize

        ans.group_ans = [GroupAnswer() for _ in range(ans.size)]

        col.sort()
        index = next((i for i, p in enumerate(col) if p[0] == groupby), -1)
        bg, ed = 0, COL_NUM[groupby]
        if in_pre:
            bg = col[index][1]
            ed = bg + 1

        # 处理 groupby
        for i in range(bg, ed):
            col[index] = (groupby, i)
            root = getRoot(col)
            queryRange(root, bound)
            _sum, _cnt = sum, cnt

            for j in range(rsize):
                idx = j if in_pre else i * rsize + j
                ans.group_ans[idx].id = i
                if res[j].op == 0:
                    ans.group_ans[idx].value = _cnt
                elif res[j].op == 1:
                    ans.group_ans[idx].value = _sum[res[j].col]
                elif res[j].op == 2:
                    ans.group_ans[idx].value = _sum[res[j].col] / cnt if cnt else 1
    else:
        ans.size = rsize
        ans.group_ans = [GroupAnswer() for _ in range(ans.size)]
        root = getRoot(col)
        queryRange(root, bound)
        _sum, _cnt = sum, cnt

        for j in range(rsize):
            ans.group_ans[j].id = -1
            if res[j].op == 0:
                ans.group_ans[j].value = _cnt
            elif res[j].op == 1:
                ans.group_ans[j].value = _sum[res[j].col]
            elif res[j].op == 2:
                ans.group_ans[j].value = _sum[res[j].col] / cnt if cnt else 1
```

```
    return ans
```