探索式数据分析大作业实验报告

数据科学导论 2021-2022 春季学期

潘俊达 2021201626 信息学院 图灵班

摘要

在探索式数据分析中，存在一类近似查询处理（AQP）问题。其旨在保证结果一定精度的情况下，通过采样、直方图、深度学习等方法，快速地统计给定限制内的数据的特征（如计数和平均值），以提高探索式数据分析的响应速度。

本次实验聚焦近似查询处理，设计了利用 KD-Tree 维护直方图，在不同资源受限情况下的具有自适应性的快速近似查询处理方法。在设计过程中，我们面临着性能，精度，资源占用，以及三者的平衡的各种挑战，大体上来说，我们利用 C 语言来提升维护数据结构的性能；通过观察查询分布来优化 KD-Tree 的分块方法，从而在同等内存的情况下提高查询的精度；最后，通过离线装载模型和在线装载相结合，平衡了时间和内存占用的限制。整个方案的代码量约 1500 行。

方法允许通过牺牲时间，牺牲内存来提高精度。经过测试，在偏向精度的参数设置下，本方法可以在 online 总时间 4.12s，offline 总时间 28s 内，完成精确查询， MSLE 1e-10 来自于浮点数误差；而偏向速度的参数设置下，在给定的 4 个 workload 中，分别依次取得了 MSLE 1.66e-15，3.01e-12，2.06e-6，1.12e-5 的较高精度，同时 online 总时间 0.738 s，offline 总时间 140 s。

1 整体方案

1.1 回顾 KD-Tree

KD-Tree 是维护多维数据的数据结构，常常被用于低维数据的范围查找等, 其能够在 $O(n^{1-\frac{1}{d}})$ 复杂度下完成一个超立方体内数据的统计。

然而在本题中，如果直接套用 KD-Tree，则最坏情况下需要大约 1.2e9 次 $O(1)$ 操作。对于一台家用机器，需要 10s 以上的时间，效率不高。这是由于 KD-Tree 的性能会随着维度的增加快速恶化[1]。

1.2 空间换时间

本实验所有查询的约束条件不大于 3 个，且对于离散型维度永远仅包含一个值。这意味着 $KD-Tree$ 中大量用来分隔的维度是浪费的。所以一种自然的想法是，我们只在查询的维度上结构从而构建 KD-Tree 即可。

[1]本题d=12，n=1000000，而 KD-Tree 一般需要 $n \gg 2^d$。

所以我们的方案是：对于共 $C_{12}^1+C_{12}^2+C_{12}^3$ 种可能的查询约束，分别构建限制高度的 KD-Tree，同时对于离散型维度，我们针对不同的取值，分别建树，这样 KD-Tree 中用于分隔的维度只可能是连续型维度，进一步减少了维度。

但这会导致空间的花费很大，我们的解决办法是：

- 在线装载：即内存存不下的 KD-Tree 会被保存在磁盘上，只有当查询到来时，才会被加载到内存中，同时随机淘汰掉之前装载进来的某个 KD-Tree。
- 折中，增大维度，减少树的数目：离散型维度的处理方法不变，但每棵树都对7个连续型维度分割，不再枚举建树，下文会比较两种方法的性能。为了区分，我们称对每个组合都单独建树的方案为维度匹配 KD-Tree，称方案2为维度冗余 KD-Tree。下文我们主要针对维度匹配 KD-Tree 展开，维度冗余 KD-Tree 的处理方法与之类似。

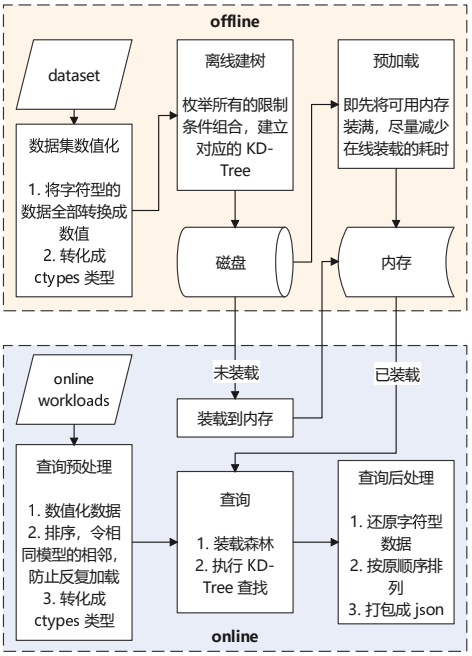

图 1: 整体方案速览

1.3 整体方案速览

图 1 展示了原始方案的整体方案的流程。其中 offline 只需要执行一次，online 会在每次调用 aqp_online 时执行。

2 详细实现

2.1 符号定义

符号	含义
r	对于某个 workload，其 predicate 涉及的维度数
r_c	r 中的连续型维度数
r_d	r 中的离散型维度数
k	指 KD-Tree 在分隔时考虑的维度数，一般情况下 $k = r_c$
m_p	KD-Tree 在分隔时，平衡分割点的下标，等于 $\frac{l+r}{2}$
m_v	KD-Tree 在分隔时，值域分割点的下标，等于 $\frac{max+min}{2}$ 在数据点中的排名
w	KD-Tree 在分隔时，加权分割点的权重，越大越倾向于平衡分割点
m_w	$m_w = wm_p + (1-w)m_v$
Forest(KD-Forest)	指一系列 KD-Tree，一个森林恰好包含一套维护整个数据集的信息
idx_{root}(树编号)	某个 KD-Tree 在其森林中的下标
bound(b)	一个超立方体，指一个 KD-Tree 节点所维护的数据边界，或查询的范围

2.2 offline 阶段

在 python 中，我们封装了 3 个主要函数，它们需要在评测的最开始时依次调用[2]：loadDataset，buildKDTrees，loadModels。

[2]之后每次调用 aqp_offline 的时候，可以可选地调用 loadModels，这会刷新当前内存中的预加载模型为预计更常见的限制情况模型，但可能导致更多的离线时间花费。

具体流程细节见图 2，流程图中尽量保留了所有有阅读价值的细节。一个森林的组织结构类似图 3，有的离散维数据取值组合没有对应的数据点所以对应的树为空树。根节点上标注了其在森林中的下标，下标和离散维数据取值一一对应。

2.2.1 离散型维度的 $O(n \log n)$ 分隔方法

本题中，离散维度的取值组合数目可能会很大，这里我们优化了这个过程。对于某组离散维度组合，我们用其对数据集进行多关键字排序。这使得具有同样取值的数据会被排列在一起，然后我们只要单调地移动左端点和右端点就可以线性地取出具有相同取值的数据点集合，从而避免了多次扫描数据集。

2.2.2 KD-Tree 构建伪代码与分割点选取问题

Algorithm 1 buildKDTree

Require: d_{max}, w, k

procedure BUILD($data, l, r, d$)
  $n = new\ Node()$
  if $r - l <= 1$ **or** $d >= d_{max}$ **or** $k = 0$ **then**
    for $i \leftarrow [l, r)$ **do**
      $n.count+ = 1$
      $n.bound = n.bound \cup data[i]$
      for $j \leftarrow [0, c)$ **do**
        $n.sum[j]+ = data[i][j]$
      end for
    end for
  else
    $axis \leftarrow depth\%k$
    $m_p \leftarrow \frac{l+r}{2}$
    $min_v \leftarrow \min(data[l:r][axis])$
    $max_v \leftarrow \max(data[l:r][axis])$
    $mid \leftarrow \frac{min_v + max_v}{2}$
    $count \leftarrow$ COUNTLESS($data[l:r][a], mid$)
    $m_v \leftarrow l + count$
    $m_w \leftarrow wm_p + (1-w)m_v$
    SPLIT($data, l, r, m_w$)
    $n.left =$BUILD($data, l, m_v, d+1$)
    $n.right =$BUILD($data, m_v, r, d+1$)
    $n.count = n.left.count + n.right.count$

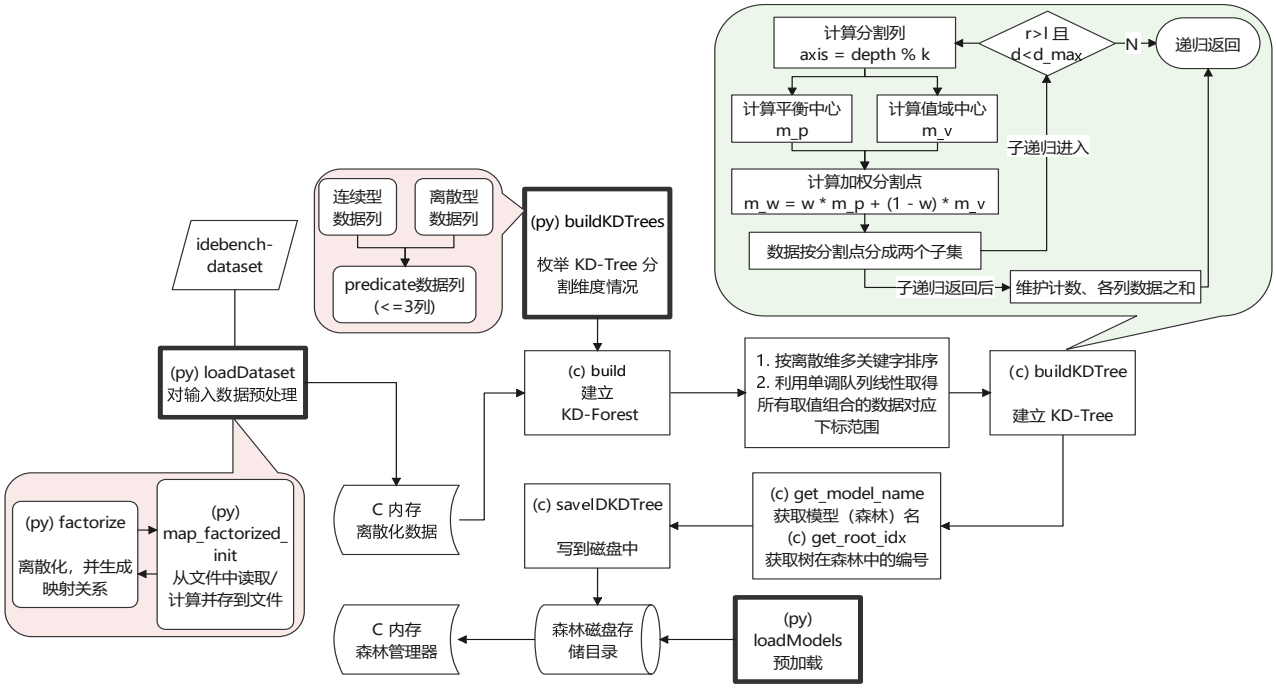

图 2: offline 阶段流程图

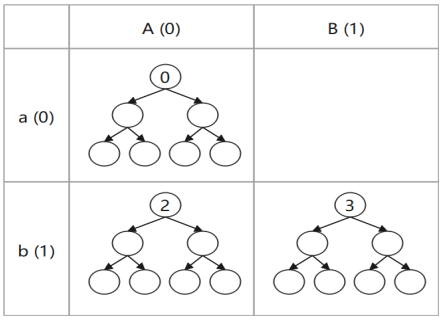

图 3: KD-Forest 示意图

24: $n.bound = n.left.bound \cup n.right.bound$
25: $n.sum = n.left.sum + n.right.sum$
26: **end if**
27: **return** n
28: **end procedure**

这里分割点的选取方法和通常的不同，通常情况下，分割点往往选取中位数，以最小化树高。

但超立方体查询算法复杂度其实比较复杂，和查询-数据分布有关，这促使我们重新审查分割的位置。

我们可以作出 workload 的查询分布图 4[3]，其反应了每个位置被查询的频率。可以发现对于连续型维度来说，数据的分布是有偏的而查询几乎是均匀的。

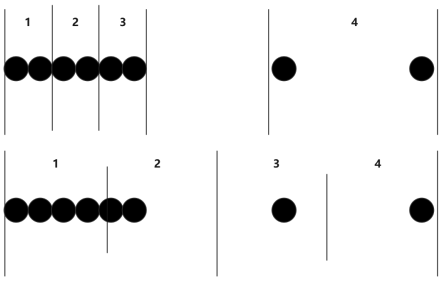

图 5: split 示意图

这意味着，如果我们按照中位数来分隔，那么就会如图 5 上半部分所示，稀疏的点被分到了一个块内，导致块内近似查询的误差很大。

而如果我们按照查询的分布来分隔，那么就会如图 5 下半部分所示，密度高的点被分到了一个块内，稀疏的点趋向于能被单独分到一个块内，从而减小了误差。

所以在考虑查询精度的情况下下，我们应该按照值域一分为二分割。

但另一方面，随着树的增高，递归的层数也会

[3]通过在查询区间内按照一定步长进行填充得到

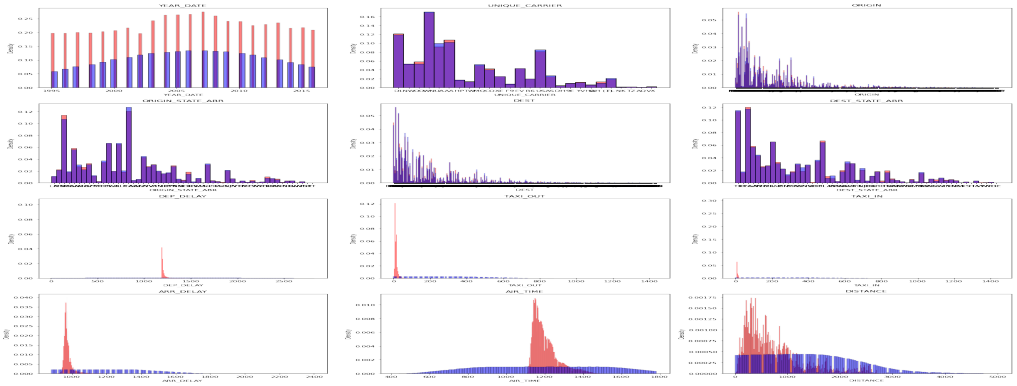

图 4: workload 查询分布

增加，而太深的递归可能导致栈溢出或额外的时间开销，同时，偏斜的树不方便确认递归边界，所以为了可控，这里采用了折中的方案，取两个中点的加权平均值作为分割点，也方便后续实验，确定哪种分隔效果好。

2.3 online 阶段

图 6 展示了 online 阶段的流程图。

2.3.1 查询伪代码

Algorithm 2 aqpQuery

1: **procedure** AQPQUERY($pred, groupBy$)
2: **if** $groupBy \neq -1$ **then**
3: **for** $v \leftarrow 0 \rightarrow valueNum_{groupBy}$ **do**
4: $pred_g \leftarrow pred \wedge groupBy = v$
5: $result[v] \leftarrow$ AQPQUERY($pred_g, -1$)
6: **end for**
7: **return** $result$
8: **else**
9: $continuePred \leftarrow pred.continuePred$
10: $discretePred \leftarrow pred.discretePred$
11: $forest \leftarrow getForest(continuePred)$
12: $idx \leftarrow getIdx(discretePred)$
13: $root \leftarrow forest[idx]$
14: $b \leftarrow getBound(continuePred)$
15: **return** KDQUERY($root, b$)
16: **end if**
17: **end procedure**
18: **procedure** KDQUERY(u, b)
Require: a is the axises that the tree is split on ▷ most likely $a = continuePred.col$
19: **if** $root = \emptyset$ **then**
20: **return** 0
21: **end if**
22: **if** $root$ is Leaf **or** $u.b[a] \subseteq u.b[a]$ **then**
23: $ratio \leftarrow \frac{\mathbb{V}(u.b \cap b)}{\mathbb{V}(u.b)}$
24: **return** $root.count \cdot ratio$ ▷ No loser, only show the maintenance of count
25: **end if**
26: $result = 0$
27: **if** $u.left.b[a] \subseteq b[a]$ **then**
28: $result$ += KDQUERY($u.left, b$)
29: **end if**
30: **if** $u.right.b[a] \subseteq b[a]$ **then**
31: $result$ += KDQUERY($u.right, b$)
32: **end if**
33: **return** $result$
34: **end procedure**

2.3.2 bound 和更多查询限制维度

本方案还考虑了现有这些树的基础上如何适应查询维度大于3的情况。

其实非常简单，只要在维护 bound 的时候，对于7个连续维度都维护：而在判断相交或包含时，只在分割维度上判断；最终统计答案时，在所有维度上统计相交比例。

图 7 展示了这种区别。在图 7 中，我们假设数据维度是 2 维，Q 是查询区间，节选了 4 个叶节

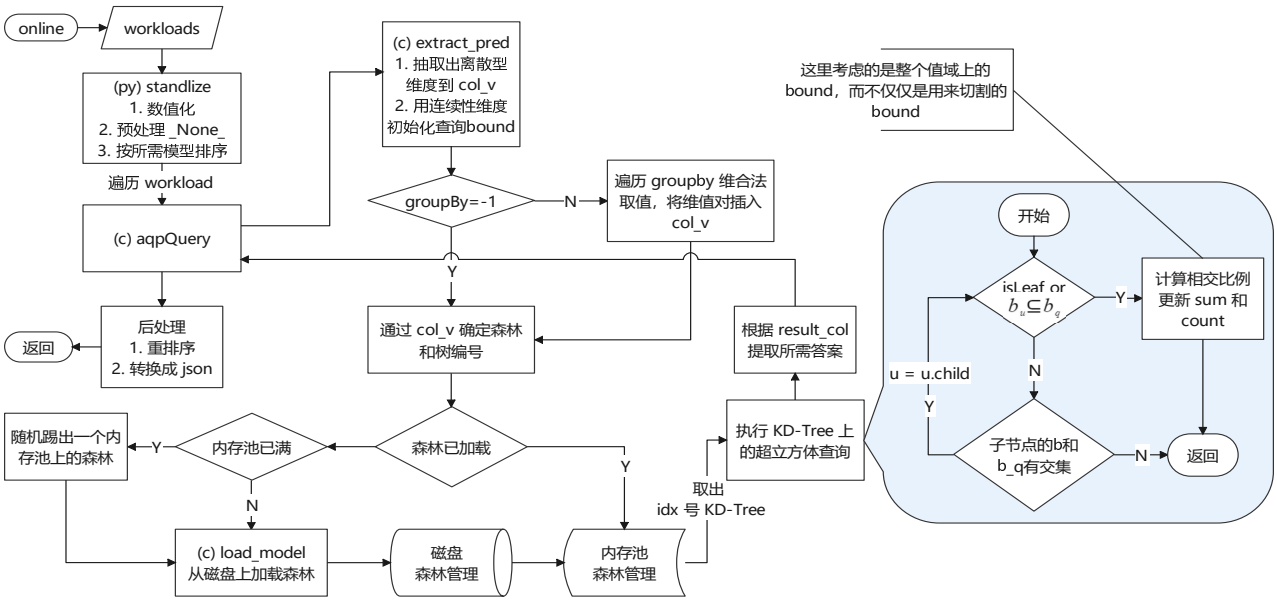

图 6: online 阶段流程图

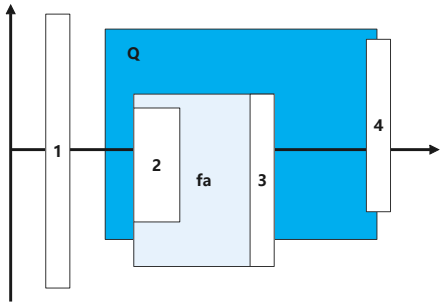

图 7: bound 示意图

点，其中 2 和 3 的父节点恰好是 fa。

对于一个在 xy 轴上都分割的 KD-Tree 来说，它最终会分别统计 2, 3, 4 对答案的贡献，然而对于一个只在 x 轴上分割的 KD-Tree 来说，当它扫描到 2 和 3 的父节点 fa 时，这个块已经被完全包含在 Q 内，从而不会继续递归下去。

可以看到后者只统计了 2 个节点，精度较差，但却节省了一次递归。综上，本方案对于大于 3 维的情况，仍然能够计算并返回有意义的答案。 [4]。

2.4 一个端到端的例子

这里我们假设数据集有4个数据维度，分别是两个连续型维度 c_1, c_2，以及离散型 d_1, d_2，我们假设所有字符型数据都已经被数值化。

为了简单，我们假设询问只可能包含至多两个维度限制。

	c_1	c_2	d_1	d_2
0	1	2	0	1
1	2	3	0	1
2	3	4	0	1
3	4	5	0	1
4	20	30	0	1
5	30	20	0	1
6	4	5	1	0
7	7	3	2	0
8	1	2	2	0

表 1: 端到端演示数据集

我们假设数据集如表 1 所示。

这里，我们演示最通用的情况，即限制条件同时包含连续和离散维度，这里不妨设为 c_1, d_1 时。

2.4.1 offline 阶段

我们此时恰好遍历到了 c_1, d_1 这种组合，现在我们针对这种组合构建一个 KD-Forest。首先我们

[4]本质上来说，KD-Tree 可以拆成直方图分块和加速统计两方面分开理解：分割维度少相当于在直方图分块的时候只从分割维度上分块，而统计的时候，我们仍然可以在所有维度上统计。

对数据按照 d_1 维进行排序。紧接着，通过单调队列扫描，我们得到 3 段取值相同的区间，分别是 $[0,6)$, $[6,7)$, $[7,9)$。通过编码，这三段区间取值对应的树编号是$0,1,2$。

这里我们演示构建 T_0，也就是 $[0,6)$ 这段区间对应的树。

我们的分割维度是连续型维，这里是 c_1，对应的取值为 $[1,2,3,4,20,30]$，这即是根节点维护的区间。

假设这里我们的分割策略是按值域分割，高度不大于 3 层，则第一个分割点的值为 $\frac{1+30}{2}$，分割后变成：$[1,2,3,4],[20,30]$。

此时两子树都未达到高度限制，区间内也有多个节点，所以还要继续分割。分割后变成：$[1,2],[3,4],[20],[30]$。对应的 index 为 $[0,1],[2,3],[4],[5]$

此时所有子树都达到了高度限制，所以停止分割，得到了 T_0，如图 8 所示。

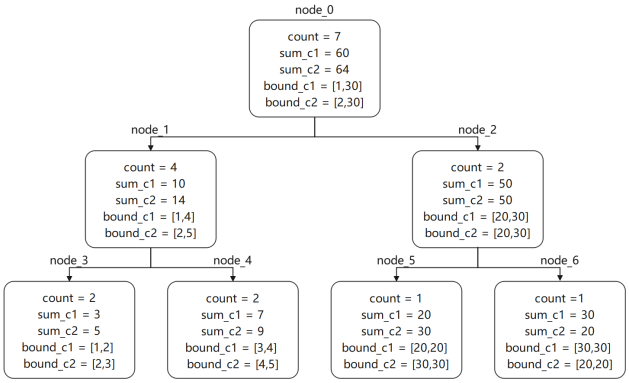

图 8: T_0 结构

紧接着，T_1 和 T_2 同理，最后得到的森林结构和图 3 类似，不再赘述。

2.4.2 online 阶段

假设 workloads 如表 2 所示。

	$result\_col$	$pred_1$	$pred_2$	$groupBy$
0	$count, *$	$c_1 \geq 3.5$	$d_1 = 0$	$\_None\_$
1	avg, c_2	$0 \leq c_1 \leq 20$	$\_None\_$	d_1

表 2: workloads

对于 $workload_0$，我们首先根据限制涉及的列 c_1, d_1 确认对应的森林，再根据 $d_1 = 0$ 确认对应森林中的 T_0 作为查找树。

查询区间为 $b = [2, \infty)$，通过 KD-Tree 算法，可以确认要统计的节点为 $node_4$ 和 $node_2$ 其中 $node_2$ 的 $bound_1$ 完全包含于 b，所以 $count+ = 2$，而 $node_4$ 的 $bound_1$ 与 b 相交，相交比例为 $\frac{0.5}{1} = 0.5$，所以 $count+ = 2 * 0.5$

综上，$count = 3$。

而对于 $workload_1$，其是 $groupBy$ 查询，所以要把 $groupBy$ 引入到条件中，遍历 $d_1 = 0, 1, 2$。

例如对于 $d_1 = 0$，现在的限制条件等价于 $0 \leq c_1 \leq 20$ 及 $d_1 = 0$。类似 $workload_0$ 可得，我们要统计的节点为 $node_1$ 和 $node_5$，计算可得 $sum = 14 + 30 = 44$, $count = 4 + 1 = 5$，所以 $avg = \frac{sum}{count} = 8.8$。 $d_1 = 1, 2$ 类似。

2.5 连续型维度完整的森林与比较

之前的分析中我们提到，我们在构建 KD-Tree 时仅在查询涉及的维度上分割，这导致了我们需要大量的树来覆盖所有组合。

这看起来是一个空间换时间的策略，然而实际测试表明，结合在值域上分割的分割策略

3 实验结果和分析

3.1 整体实验结果

在 8C 16G 的 dlab 环境上测试，实验结果如表 3, 4 所示，其中模型的构建仅在第一次调用 offline 时执行并统计，其余的 offline 事实上只做了森林的重新加载工作。

我们给出了两个方法的对比，前者是维度匹配，即建小而精细的大量树，它可以在很短的时间内完成 aqp 任务，同时内存和时间可控（即使内存只有1G也可以取得相同数量级的精度）；后者是维度冗余，其精度只在满树时优于维度匹配（即其只在求精确值时优于维度匹配），所以时间效率较差。这里，因为给的内存很大，所以都没有出现需要在线加载的问题，但这是很重要的部分，所以下一节我们会讨论这一问题。

workload	MSLE	online (s)	offline (s)
Q1	1.66e-15	0.149	139
Q2	3.01e-12	0.161	0
Q3	2.06e-06	0.149	0
Q4	1.12e-05	0.290	0
MEAN	3.25e-06	-	-
SUM	-	0.738	139

· 峰值内存 5.6 G

表 3: 维度匹配 KD-Tree 实验结果

workload	MSLE	online (s)	offline (s)
Q1	0	0.73	27.1
Q2	8.95e-29$^+$	0.93	0
Q3	4.46e-29$^+$	0.96	0
Q4	4.01e-29$^+$	1.34	0
MEAN	4.12e-29$^+$	-	-
SUM	-	4.16	27.1

$^+$ 表示有误差，但误差是由于浮点数精度导致的

· 峰值内存 6.1 GB

表 4: 维度冗余 KD-Tree 实验结果

3.2 参数分析

3.2.1 分割策略

这一节，我们测试了分割权重变化对查询精度的影响，如图 9 和 10 所示。

结果显示，对于维度匹配 KD-Tree 来说，在值域上分割能在降低内存的同时显著提升精度。然而，维度冗余 KD-Tree 则不然。维度冗余 KD-Tree 的最优权重随着树的增高慢慢往 1（平分）靠近。

猜测有两种可能，分别对应了两种实验现象：其一是观察 $deltaDelth \leq 1$ 时，对于这样深的树来说，其精度主要取决于是不是准确统计了每个点，所以用于优化近似的策略无效了；其二是同样树高下，高维 KD-Tree 的精度远低于低维，可能在值域上分割非查询的维度打乱了数据在查询维度的排列，破坏了这个优化的基础：数据的聚集性。

3.3 精度和时间，内存的关系

本方案中，决定精度、时间和内存的主要因素是树的高度，测试结果如图 11 所示。

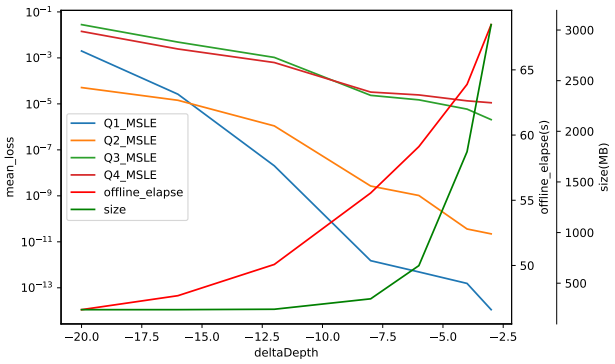

图 11: 不同树高下的精度、离线时间和内存的关系

可以看到，随着树的增高，精度、离线建树时间和内存的增长都符合指数增长的趋势。

但比较特别的是在线时间，因为我们采用了高效的 KD-Tree 算法，所以实际用于查询的时间是很短的，在假设内存不受限的情况下，结果如图 12 所示，在线时间的变化是比较小的。

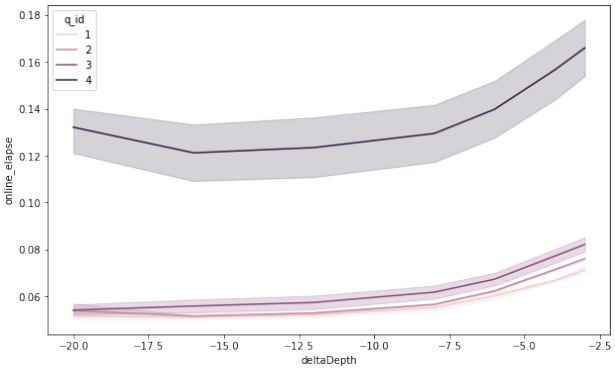

图 12: 不同树高下的在线时间（内存充裕）

但内存不足时，模型无法在离线阶段就全部装载进内存，这意味着在线时间会包含十分耗时的 IO 操作。

为了突出这一点，我们假设内存的限制为 512MB 左右，此时的统计结果如图 13 所示。

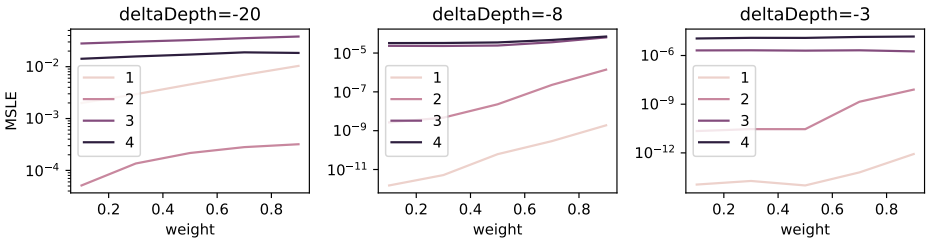

图 9: 分割策略对维度匹配 KD-Tree 上查询精度的影响

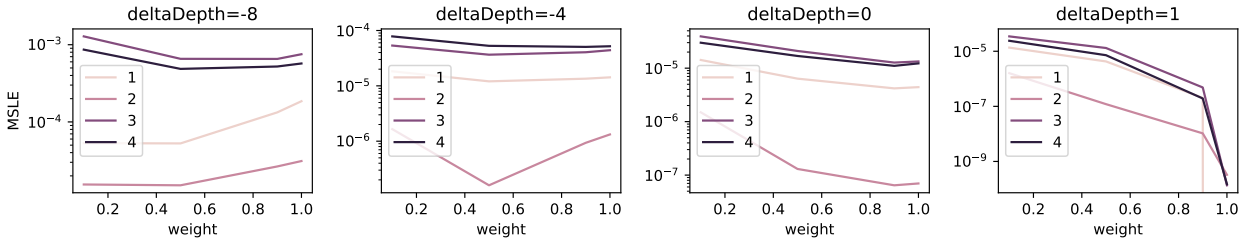

图 10: 分割策略对维度冗余 KD-Tree 上查询精度的影响

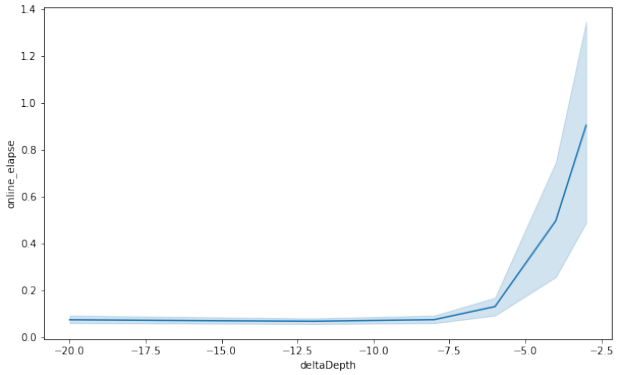

图 13: 不同树高下的在线时间（内存受限）

4 总结与体会

写完之前没想到这套方法的代码量这么大（这么卷）。起因是按照 C 的效率分析了一下，理论上 5s 都够做准确计算了。

本次实验某种程度上也是对类似 numpy 这样的库是如何将 C 嵌入到 Python 中的一次探索。第一次大规模地试验了一下这种开发方式，也算挺有意义。

附录

.1 源代码

由于代码量大约有 1500 行，所以在这里阅读并不是一个好选择，建议访问 `https://github.com/panjd123/kdtree-aqp` 查看。最终提交的代码可能会有一些参数上的差异。

1. aqp.py: 负责 json 解析，调用 kdtreeaqp.py 的接口完成模型初始化和查询
2. kdtreeaqp.py：负责标准化数据集和查询，从而生成能传入 C处理的数据流，最终与模型直接交互的部分是调用 libaqp.so 的接口
3. libaqp.h：头文件，定义了 libaqp.so 的接口
4. libaqp.cc：实现文件，具体实现了 libaqp.so 的接口，比如 KD-Tree 算法，内存的动态载入等

Listing 1: aqp.py

```python
import pandas as pd
import json
from tqdm import tqdm
import sys
import os.path as osp
import numpy as np

lib_dir = osp.join(osp.dirname(osp.abspath(__file__)), "codes")
sys.path.append(lib_dir)
import kdtree_aqp as aqplib


def aqp_online(data: pd.DataFrame, Q: list) -> list:
    workloads = pd.json_normalize([json.loads(i) for i in Q])

    workloads, arg = aqplib.standlize(workloads)

    results = []

    for _, workload in tqdm(workloads.iterrows()):
        ans = aqplib.query(workload)
        results.append(ans)

    arg = np.argsort(arg)
    results = [results[i] for i in arg]

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

```

```
29      return results
30
31
32  def aqp_offline(data: pd.DataFrame, Q: list) -> None:
33      aqplib.loadDataset(data)
34      aqplib.buildKDTrees(force=False)
35      aqplib.loadModels()
```

Listing 2: kdtreeaqp.py

```
1   import ctypes
2   from ctypes import CDLL, POINTER, Structure, c_int, c_float
3   import numpy as np
4   import pandas as pd
5   import os.path as osp
6   import os
7   from timeit import default_timer as timer
8   import atexit
9   from itertools import combinations, product
10  from tqdm import tqdm
11  from scipy.special import comb
12
13  WORKING_DIR = osp.dirname(osp.dirname(osp.abspath(__file__)))
14  # WORKING_DIR = '*/2021201626/'
15  HANDOUTS_DIR = osp.dirname(WORKING_DIR)
16  DATA_DIR = osp.join(WORKING_DIR, 'tmp') # Important!
17  MODEL_DIR = osp.join(DATA_DIR, 'models')
18
19  if not osp.exists(DATA_DIR):
20      os.mkdir(DATA_DIR)
21
22  CODE_DIR = osp.join(WORKING_DIR, 'codes')
23  DATASET_DIR = osp.join(HANDOUTS_DIR, 'data')
24  WORKLOAD_DIR = osp.join(HANDOUTS_DIR, 'workload')
25
26  TINY_DATASET_PATH = osp.join(DATASET_DIR, 'idebench-dataset-tiny.csv
        ')
27  DATASET_PATH = osp.join(DATASET_DIR, 'idebench-dataset.csv')
28
29  COLUMNS = ['YEAR_DATE', 'DEP_DELAY', 'TAXI_OUT', 'TAXI_IN', '
        ARR_DELAY',
```

```
30              'AIR_TIME', 'DISTANCE', 'UNIQUE_CARRIER', 'ORIGIN', '
                    ORIGIN_STATE_ABR', 'DEST',
31              'DEST_STATE_ABR']
32  DISCRETE_COLUMNS = ['UNIQUE_CARRIER', 'ORIGIN', 'ORIGIN_STATE_ABR',
        'DEST',
33                          'DEST_STATE_ABR']
34  CONTINUOUS_COLUMNS = ['YEAR_DATE', 'DEP_DELAY', 'TAXI_OUT', 'TAXI_IN
        ', 'ARR_DELAY',
35                            'AIR_TIME', 'DISTANCE']
36  Q_ID_LIST = ['Q1', 'Q2', 'Q3', 'Q4']
37
38  COLUMN2INDEX = {c: i for i, c in enumerate(COLUMNS)}
39  COLUMN2INDEX.update({"*":-1})
40  COLUMN2INDEX.update({"_None_":-1})
41  INDEX2COLUMN = COLUMNS
42
43  DATASET, VALUE2ID, ID2VALUE = None, None, None
44
45  MODE = 'performance' # 'memory' or 'performance'
46
47  def get_dataset():
48      df = pd.read_csv(DATASET_PATH)
49      df = df[COLUMNS]
50      return df
51
52  def get_workload_path(q_id='tiny'):
53      '''offline, online'''
54      offline_workload_path = osp.join(WORKLOAD_DIR,
55                                       '{}-workload-{}.json'.format('
                                            offline',q_id))
56      online_workload_path = osp.join(WORKLOAD_DIR,
57                                      '{}-workload-{}.json'.format('
                                           online',q_id))
58      return offline_workload_path, online_workload_path
59
60  def get_workloads(online=True, offline=True, q_id_list=Q_ID_LIST):
61      assert online or offline
62      workloads = []
63      for q_id in q_id_list:
64          offline_workload_path, online_workload_path =
                get_workload_path(q_id)
```

```
65          if online:
66              online_workloads = pd.read_json(online_workload_path,
                    orient='records', lines=True)
67              workloads.append(online_workloads)
68          if offline:
69              offline_workloads = pd.read_json(offline_workload_path,
                    orient='records', lines=True)
70              workloads.append(offline_workloads)
71      return pd.concat(workloads, ignore_index=True)
72
73  def factorize(df, columns=DISCRETE_COLUMNS):
74      id2value = {}
75      value2id = {}
76      df = df.copy()
77      for col in columns:
78          values, index = pd.factorize(df[col])
79          df[col] = values
80          col_id = COLUMN2INDEX[col]
81          id2value.update({col_id: index.values})
82          value2id.update({col_id: {v: i for i, v in enumerate(index)
                }})
83          id2value.update({col: index.values})
84          value2id.update({col: {v: i for i, v in enumerate(index)}})
85
86      return df, id2value, value2id
87
88  def get_modelName(predicate, groupBy)->str:
89      col = [int(p[0]) for p in predicate]
90      if groupBy not in col and groupBy != -1:
91          col.append(groupBy)
92      col = sorted(col)
93      modelName = '_'.join([str(c) for c in col])
94      return modelName
95
96  def standlize(workloads:pd.DataFrame):
97      '''
98      Numerical and sorting,
99      sorting is to allow the queries of the same model to be adjacent
            to and avoid repeated loading models.
100     '''
101
```

```
    # result_col
    op_map = {'count': 0, 'sum':1, 'avg':2}
    standlized_result_col = []
    for result_col in workloads['result_col']:
        standlized_result_col.append(
            [(op_map[i[0]],COLUMN2INDEX[i[1]]) for i in result_col
                if len(i) >= 2])
    workloads['result_col'] = standlized_result_col

    # predicate
    standlized_predicate = []
    for predicate in workloads['predicate']:
        predicate_tmp = []
        for i in predicate:
            col = COLUMN2INDEX[i['col']]
            if i['col'] in DISCRETE_COLUMNS:
                lb = ub = VALUE2ID[i['col']][i['lb']]
            else:
                if i['lb'] == '_None_':
                    lb = 0
                else:
                    lb = float(i['lb'])
                if i['ub'] == '_None_':
                    ub = 1e9
                else:
                    ub = float(i['ub'])
            predicate_tmp.append((col,lb,ub))
        standlized_predicate.append(predicate_tmp)
    workloads['predicate'] = standlized_predicate

    # groupby
    standlized_groupby = workloads['groupby'].apply(COLUMN2INDEX.get
        )
    workloads['groupby'] = standlized_groupby

    # model
    workloads['model'] = workloads.apply(lambda x: get_modelName(x['
        predicate'], x['groupby']), axis=1)


    # sort
```

```
140     arg = workloads['model'].argsort()
141     workloads = workloads.iloc[arg, :]
142 
143     return workloads, arg
144 
145 def _get_valueMaps_from_dataset(dataset):
146     dataset, id2value, value2id = factorize(dataset)
147     maps = [value2id,id2value]
148     np.save(osp.join(DATA_DIR,'valueMaps.npy'), maps)
149     dataset.to_csv(osp.join(DATA_DIR,'factorizedDataset.csv'), index
        =False)
150     return dataset, value2id, id2value
151 
152 def map_factorized_init(dataset=None):
153     global DATASET, VALUE2ID, ID2VALUE
154     if osp.exists(osp.join(DATA_DIR,'valueMaps.npy')) and osp.exists
        (osp.join(DATA_DIR,'factorizedDataset.csv')):
155         DATASET = pd.read_csv(osp.join(DATA_DIR,'factorizedDataset.
            csv'))
156         VALUE2ID, ID2VALUE = np.load(osp.join(DATA_DIR,'valueMaps.
            npy'), allow_pickle=True)
157     elif dataset is not None:
158         DATASET, VALUE2ID, ID2VALUE = _get_valueMaps_from_dataset(
            dataset)
159 
160 lib = CDLL(osp.join(CODE_DIR,'libaqp.so'))
161 
162 class Operation(Structure):
163     _fields_ = [('col', c_int), ('op', c_int)]
164 
165 class Predication(Structure):
166     _fields_ = [('col', c_int), ('lb', c_float), ('ub', c_float)]
167 
168 class GroupAnswer(Structure):
169     _fields_ = [('id', c_int), ('value', c_float)]
170 
171 class Answer(Structure):
172     _fields_ = [('group_ans', POINTER(GroupAnswer)), ('size', c_int)
        ]
173 
174 class AnswerBatch(Structure):
```

```
175     _fields_ = [('ans', POINTER(Answer)), ('size', c_int)]
176
177 def loadDataset(dataset):
178     dataset = dataset[COLUMNS]
179     map_factorized_init(dataset)
180     dataset = DATASET.copy()
181     values = dataset.values.astype(np.float32)
182     values = values.flatten()
183     lib.loadData(values.ctypes.data_as(POINTER(c_float)), dataset.
            shape[0])
184
185 def buildKDTrees(force=True):
186     if not osp.exists(MODEL_DIR):
187         os.mkdir(MODEL_DIR)
188     mode = MODE
189     if not force and osp.exists(osp.join(MODEL_DIR, 'model_list.txt'
            )):
190             return
191     if osp.exists(osp.join(MODEL_DIR, 'model_list.txt')):
192         os.remove(osp.join(MODEL_DIR, 'model_list.txt'))
193     if mode == 'performance':
194         for pred_num in [1, 2, 3]:
195             for col in tqdm(combinations(range(0,12), pred_num),
                    total=int(comb(12, pred_num))):
196                 col_np = np.array(col, dtype=np.int32)
197                 lib.build(col_np.ctypes.data_as(POINTER(c_int)), len
                        (col_np), -6, 0.1)
198     else: # mode == 'memory'
199         for di_pred_num in [0, 1, 2, 3]:
200             for di_col in tqdm(combinations(range(7,12), di_pred_num
                    ), total=int(comb(5, di_pred_num))):
201                 if di_pred_num < 3:
202                     col_np = np.r_[np.array(range(7), dtype=np.int32
                            ),
203                                    np.array(di_col, dtype=np.int32)]
204                 else:
205                     col_np = np.array(di_col, dtype=np.int32)
206                 lib.build(col_np.ctypes.data_as(POINTER(c_int)), len
                        (col_np), 0, 1)
207
208 def query(workload):
```

```
209     if MODE=='performance':
210         mode = 0
211     else: # mode == 'memory'
212         mode = 1
213 
214     ops = np.array(workload['result_col'], dtype=Operation)
215     preds = np.array(workload['predicate'], dtype=Predication)
216     groupBy_col = workload['groupby']
217     # print(ops, preds, groupBy_col, workload['ground_truth'])
218     ans = lib.aqpQuery(ops.ctypes.data_as(POINTER(Operation)),
219                        len(ops),
220                        preds.ctypes.data_as(POINTER(Predication)),
221                        len(preds),
222                        groupBy_col,
223                        mode)
224     ans = ans.contents
225     size = ans.size
226     ret = []
227     for i in range(size):
228         g_ans = ans.group_ans[i]
229         if g_ans.id < 0:
230             ret.append([g_ans.value])
231         else:
232             id = g_ans.id
233             ret.append([ID2VALUE[groupBy_col][id], g_ans.value])
234     return ret
235 
236 @atexit.register
237 def clear():
238     lib.clear()
239 
240 def lib_init():
241     lib.aqpQuery.argtypes = [POINTER(Operation), c_int, POINTER(
        Predication), c_int, c_int, c_int]
242     lib.aqpQuery.restype = POINTER(Answer)
243 
244     lib.loadData.argtypes = [POINTER(c_float), c_int]
245     lib.loadData.restype = None
246 
247     lib.clear.argtypes = []
248     lib.clear.restype = None
```

```
249
250         lib.build.argtypes = [POINTER(c_int), c_int, c_int, c_float]
251         lib.build.restype = None
252
253         lib.init.argtypes = [ctypes.c_char_p]
254         lib.init.restype = None
255         dir = MODEL_DIR
256         lib.init(dir.encode('utf-8'))
257
258         map_factorized_init(None)
259
260     def loadModels():
261         lib.load_models()
262
263     lib_init()
```

Listing 3: libaqp.h

```
1  #pragma once
2  #include <algorithm>
3  #include <array>
4  #include <cstdio>
5  #include <cstdlib>
6  #include <utility>
7  #include <vector>
8
9  const int DATA_DIM = 7;
10 const int COL_NUM = 12;
11 using COL_T = int;
12 enum OP {
13     COUNT,
14     SUM,
15     AVG,
16 };
17 using OP_T = OP;
18 using INT_T = int;
19 using FLOAT_T = float;
20 // using DATA_T = FLOAT_T[DATA_DIM];
21 using DATA_T = std::array<FLOAT_T, DATA_DIM>;
22 using BOUND_T = float[DATA_DIM][2];
23 using COL_VALUE_T = std::vector<std::pair<int, int>>;
```

```
24
25 #define IS_LEAF(u) ((u)->lchild == nullptr && (u)->rchild == nullptr
       )
26 #define IS_CONTINUED(c) ((c <= 6))
27 #define IS_DISCRETE(c) ((c >= 7))
28 #define GB (1024ull * 1024 * 1024)
29 #define MEM_LIMIT (0.5 * GB)
30
31 enum MODE {
32     PERFORMANCE,
33     MEMORY,
34 };
35
36 struct Operation {
37     OP_T op;
38     COL_T col;
39 };
40
41 struct Predication {
42     COL_T col;
43     FLOAT_T lb;
44     FLOAT_T ub;
45 };
46
47 struct GroupAnswer {
48     INT_T id;
49     FLOAT_T value;
50 };
51
52 struct Answer {
53     GroupAnswer* group_ans;
54     int size;
55 };
56
57 struct AnswerBatch {
58     Answer* ans;
59     int size;
60 };
61
62 struct Node {
63     struct Node* lchild;
```

```
64      struct Node* rchild;
65      int count;
66      FLOAT_T sum[DATA_DIM];
67      BOUND_T bound;
68  };
69
70  /* KD tree module */
71
72  // Building a tree based on the basic parameters of the KD tree
73  extern "C" void build(INT_T* col, int size, int delta_depth, float
        _build_k);
74
75  // Calculate the cross ratio for approximate calculations (consider
        all dimensions)
76  double data_cross_ratio(const BOUND_T& bound_in, const BOUND_T&
        bound_out);
77
78  // Whether it includes (only considering the KD tree segmentation
        dimension)
79  int kd_contain(const BOUND_T& bound_in, const BOUND_T& bound_out);
80
81  // Whether to intersect (considering only the KD tree segmentation
        dimension)
82  int kd_cross(const BOUND_T& bound_in, const BOUND_T& bound_out);
83
84  // Use the data in the range of [L, R) to establish a KD tree
85  Node* buildKDTree(DATA_T* data, int l, int r, int depth);
86
87  // Write tree to hard disk
88  void saveKDTree(Node* u);
89
90  // Write a ID before writing a tree
91  void saveIDKDTree(Node* u, int id);
92
93  // Load a tree to memory
94  Node* loadKDTree();
95
96  // Release tree memory
97  void clearKDTree(Node* u);
98
99  void testKDTree(Node* u, int depth);
```

```
100
101 // Print tree to the screen
102 void printKDTree(Node* u, int depth);
103
104 /* Query module */
105
106 // Get the model name corresponding to the column
107 std::string get_model_name(COL_VALUE_T& col_value);
108
109 // Get the model path corresponding to model_name
110 std::string get_model_path(std::string model_name);
111
112 // Get the root node of the KD tree corresponding to the column
113 Node* get_root(COL_VALUE_T col_value);
114
115 // Recursive query
116 void _queryRange(Node* u, const BOUND_T& bound, FLOAT_T* sum, double
      * count);
117
118 // Initialization query and recursive query
119 void queryRange(Node* root, BOUND_T& bound, FLOAT_T* sum, double&
      count);
120
121 // Get the answer to the query
122 extern "C" Answer* aqpQuery(Operation* ops, int, Predication* preds,
       int, COL_T, MODE);
123
124 // Release the memory of the last answer
125 void clearans();
126
127 /* Initialization module */
128
129 void load_col_type();
130
131 extern "C" void init(const char* dir);
132
133 /* Load dataset module */
134
135 extern "C" void loadData(FLOAT_T* _data, int n);
136
137 void clearData();
```

```
138
139 /* Loading model */
140
141 void load_model(const std::string model_name);
142
143 void clear_model(const std::string model_name);
144
145 extern "C" void load_models();
146
147 /* Destructive module */
148
149 void clear_models();
150
151 extern "C" void clear();
```

Listing 4: libaqp.cc

```
1  #include "libaqp.h"
2  #include <algorithm>
3  #include <array>
4  #include <cmath>
5  #include <cstdio>
6  #include <cstdlib>
7  #include <cstring>
8  #include <queue>
9  #include <unordered_map>
10 #include <utility>
11 #include <vector>
12
13 static bool is_init = false;
14
15 /**** dataset ****/
16
17 static FLOAT_T* dataset = nullptr;
18 static int dataset_size;
19 static DATA_T* data = nullptr;
20 static int& data_size = dataset_size;
21
22 static size_t working_memory = 0;
23 static size_t max_working_memory = 0;
24 static size_t total_memory = 0;
```

```
25
26  /**** Build KD-Tree ****/
27
28  static COL_T split_axises[12];
29  static int split_axis_num;
30  static int max_depth = 20;
31  static FILE* model_file;
32
33  // k larger, performance better
34  // k smaller, accuracy better
35  static float build_k = 0.1;
36
37  double data_cross_ratio(const BOUND_T& bound_in, const BOUND_T&
        bound_out) {
38      double ratio = 1;
39      for (int i = 0; i < DATA_DIM; i++) {
40          if (bound_in[i][0] == bound_in[i][1]) {
41              ratio *= (bound_out[i][0] <= bound_in[i][0] && bound_in[
                    i][0] <= bound_out[i][1]);
42          } else {
43              double l, r;
44              l = std::max(bound_in[i][0], bound_out[i][0]);
45              r = std::min(bound_in[i][1], bound_out[i][1]);
46              ratio *= (r - l) / (bound_in[i][1] - bound_in[i][0]);
47          }
48      }
49      return ratio;
50  }
51
52  int kd_contain(const BOUND_T& bound_in, const BOUND_T& bound_out) {
53      for (int i = 0; i < split_axis_num; i++) {
54          int split_axis = split_axises[i];
55          if (bound_in[split_axis][0] < bound_out[split_axis][0] ||
56              bound_in[split_axis][1] > bound_out[split_axis][1]) {
57              return 0;
58          }
59      }
60      return 1;
61  }
62
63  int kd_cross(const BOUND_T& bound_in, const BOUND_T& bound_out) {
```

```
64      for (int i = 0; i < split_axis_num; i++) {
65          int split_axis = split_axises[i];
66          if (bound_in[split_axis][0] > bound_out[split_axis][1] ||
67              bound_in[split_axis][1] < bound_out[split_axis][0]) {
68              return 0;
69          }
70      }
71      return 1;
72  }
73
74  Node* buildKDTree(DATA_T* data, int l, int r, int depth) {
75      if (l > r) {
76          return nullptr;
77      }
78      Node* u = new Node;
79
80      if (l == r || depth >= max_depth || split_axis_num == 0) {
81          u->lchild = u->rchild = nullptr;
82          // u->split_value = 0;
83          u->count = r - l + 1;
84          for (int i = 0; i < DATA_DIM; i++) {
85              u->sum[i] = 0;
86              u->bound[i][0] = 1e9;
87              u->bound[i][1] = -1e9;
88              for (int j = l; j <= r; j++) {
89                  u->sum[i] += data[j][i];
90                  u->bound[i][0] = std::min(u->bound[i][0], data[j][i
                        ]);
91                  u->bound[i][1] = std::max(u->bound[i][1], data[j][i
                        ]);
92              }
93          }
94          return u;
95      }
96
97      COL_T split_dim = split_axises[depth % split_axis_num];
98      // std::nth_element(data + l, data + (l + r) / 2, data + r + 1,
99      //                  [split_dim](const DATA_T& a, const DATA_T& b
            ) {
100     //                      return a[split_dim] < b[split_dim];
101     //                  });
```

```
102     int perfomance_median = (l + r) / 2;
103
104     /* get the position of the mid value */
105     int accuracy_median = l;
106
107     if (build_k != 1) {
108         FLOAT_T min(1e9), max(-1e9), mid_value;
109         for (int i = l; i <= r; i++) {
110             if (data[i][split_dim] < min) {
111                 min = data[i][split_dim];
112             }
113             if (data[i][split_dim] > max) {
114                 max = data[i][split_dim];
115             }
116         }
117         mid_value = (min + max) / 2;
118         for (int i = l; i <= r; i++) {
119             if (data[i][split_dim] < mid_value) {
120                 accuracy_median++;
121             }
122         }
123     }
124
125     // consider both performance and accuracy
126     int median = perfomance_median * build_k + accuracy_median * (1
        - build_k);
127
128 #ifdef INFO
129     printf("%.2f %.2f %.2f\n", min, mid_value, max);
130     printf("perfomance_median: %d, accuracy_median: %d, median: %d\n
        ", perfomance_median, accuracy_median, median);
131 #endif
132
133     if (median == r) {
134         u->lchild = u->rchild = nullptr;
135         // u->split_value = 0;
136         u->count = r - l + 1;
137         for (int i = 0; i < DATA_DIM; i++) {
138             u->sum[i] = 0;
139             u->bound[i][0] = 1e9;
140             u->bound[i][1] = -1e9;
```

```
141             for (int j = l; j <= r; j++) {
142                 u->sum[i] += data[j][i];
143                 u->bound[i][0] = std::min(u->bound[i][0], data[j][i
                        ]);
144                 u->bound[i][1] = std::max(u->bound[i][1], data[j][i
                        ]);
145             }
146         }
147         return u;
148     }
149 
150     std::nth_element(data + l, data + median, data + r + 1,
151                      [split_dim](const DATA_T& a, const DATA_T& b) {
152                          return a[split_dim] < b[split_dim];
153                      });
154 
155     // u->split_value = data[median][split_dim];
156     u->count = r - l + 1;
157 
158     // [l, median]
159     u->lchild = buildKDTree(data, l, median, depth + 1);
160     // [median + 1, r]
161     u->rchild = buildKDTree(data, median + 1, r, depth + 1);
162 
163     for (int i = 0; i < DATA_DIM; i++) {
164         u->sum[i] = 0;
165         u->bound[i][0] = 1e9;
166         u->bound[i][1] = -1e9;
167         if (u->lchild != nullptr) {
168             u->sum[i] += u->lchild->sum[i];
169             u->bound[i][0] = std::min(u->bound[i][0], u->lchild->
                    bound[i][0]);
170             u->bound[i][1] = std::max(u->bound[i][1], u->lchild->
                    bound[i][1]);
171         }
172         if (u->rchild != nullptr) {
173             u->sum[i] += u->rchild->sum[i];
174             u->bound[i][0] = std::min(u->bound[i][0], u->rchild->
                    bound[i][0]);
175             u->bound[i][1] = std::max(u->bound[i][1], u->rchild->
                    bound[i][1]);
```

```
176             }
177         }
178 
179         return u;
180     }
181 
182     void _queryRange(Node* u, const BOUND_T& bound, FLOAT_T* sum, double
        & count) {
183         if (u == nullptr) {
184             return;
185         }
186         // for (int i = 0; i < DATA_DIM; i++) {
187         //     printf("%.2f %.2f\n", u->bound[i][0], u->bound[i][1]);
188         // }
189         if (IS_LEAF(u) || kd_contain(u->bound, bound)) {
190             double ratio = data_cross_ratio(u->bound, bound);
191             count += u->count * ratio;
192             for (int i = 0; i < DATA_DIM; i++) {
193                 sum[i] += u->sum[i] * ratio;
194             }
195             return;
196         }
197         if (u->lchild && kd_cross(u->lchild->bound, bound)) {
198             _queryRange(u->lchild, bound, sum, count);
199         }
200         if (u->rchild && kd_cross(u->rchild->bound, bound)) {
201             _queryRange(u->rchild, bound, sum, count);
202         }
203     }
204 
205     void saveKDTree(Node* u) {
206         if (!model_file || !u) {
207             return;
208         }
209         fwrite(u, sizeof(Node), 1, model_file);
210         saveKDTree(u->lchild);
211         saveKDTree(u->rchild);
212     }
213 
214     void saveIDKDTree(Node* u, int id) {
215         if (!model_file || !u) {
```

```
216             return;
217         }
218         fwrite(&id, sizeof(int), 1, model_file);
219         saveKDTree(u);
220     }
221
222     Node* loadKDTree() {
223         Node* u = new Node;
224         working_memory += sizeof(Node);
225         size_t _ = fread(u, sizeof(Node), 1, model_file);
226         if (_ < 1) {
227             printf("loadKDTree error\n");
228             return nullptr;
229         }
230         if (u->lchild != nullptr) {
231             u->lchild = loadKDTree();
232         }
233         if (u->rchild != nullptr) {
234             u->rchild = loadKDTree();
235         }
236         return u;
237     }
238
239     void clearKDTree(Node* u) {
240         if (u == nullptr) {
241             return;
242         }
243         working_memory += sizeof(Node);
244         clearKDTree(u->lchild);
245         clearKDTree(u->rchild);
246         delete u;
247     }
248
249     /**** Query Function ****/
250
251     // >=0 for continuous, -1 for discrete
252     static int col_map[COL_NUM];
253     static int value_num[COL_NUM];
254     static std::unordered_map<std::string, std::unordered_map<int, Node
        *>> model_map;
255     static std::vector<std::string> model_list;
```

```
256
257 std::string get_model_name(COL_VALUE_T& col_value) {
258     std::string model_name = "";
259     for (int i = 0; i < col_value.size(); i++) {
260         model_name += std::to_string(col_value[i].first);
261         // if (col_value[i].second >= 0) {
262         //     model_name += "=" + std::to_string(col_value[i].
                second);
263         // }
264         if (i != col_value.size() - 1) {
265             model_name += "_";
266         }
267     }
268     return model_name;
269 }
270
271 Node* get_root(COL_VALUE_T col_value) {
272     int size = col_value.size();
273     int root_idx = 0;
274     int tmp = 1;
275     for (int i = 0; i < size; i++) {
276         int col = col_value[i].first;
277         int value = col_value[i].second;
278         if (value >= 0) {
279             root_idx += tmp * value;
280             tmp *= value_num[col];
281         }
282     }
283     std::string model_name = get_model_name(col_value);
284     if (model_map.find(model_name) == model_map.end()) {
285 // printf("model %s not found\n", model_name.c_str());
286 #if 0
287         printf("load_%s\n", model_name.c_str());
288 #endif
289         load_model(model_name);
290     }
291     Node* root = model_map[model_name][root_idx];
292     return root;
293 }
294
295 void load_col_type() {
```

```
296     // FILE* col_file = fopen("col.txt", "rb");
297     // fread(col_map, sizeof(int), COL_NUM, col_file);
298     // fread(value_num, sizeof(int), COL_NUM, col_file);
299     for (int i = 0; i < 7; i++)
300         col_map[i] = i;
301     for (int i = 7; i < COL_NUM; i++)
302         col_map[i] = -1;
303     for (int i = 0; i < 7; i++)
304         value_num[i] = 1;
305     value_num[7] = 26;
306     value_num[8] = 363;
307     value_num[9] = 53;
308     value_num[10] = 366;
309     value_num[11] = 53;
310 }
311
312 static std::string MODEL_DIR;
313
314 std::string get_model_path(std::string model_name) {
315     return MODEL_DIR + "/model_" + model_name + ".bin";
316 }
317
318 void load_model(const std::string model_name) {
319     if (model_map.find(model_name) != model_map.end()) {
320         return;
321     }
322     while (total_memory > MEM_LIMIT && model_list.size() > 0) {
323         int rd = rand() % model_list.size();
324         clear_model(model_list[rd]);
325         model_list.erase(model_list.begin() + rd);
326     }
327     std::string model_path = get_model_path(model_name);
328     model_file = fopen(model_path.c_str(), "rb");
329     model_map[model_name] = std::unordered_map<int, Node*>();
330     int idx;
331     working_memory = 0;
332     while (fread(&idx, sizeof(int), 1, model_file) == 1) {
333         model_map[model_name][idx] = loadKDTree();
334     }
335     max_working_memory = std::max(max_working_memory, working_memory
        );
```

```
336     total_memory += working_memory;
337 }
338
339 void clear_model(const std::string model_name) {
340     if (model_map.find(model_name) == model_map.end()) {
341         return;
342     }
343     working_memory = 0;
344     for (auto& it : model_map[model_name]) {
345         clearKDTree(it.second);
346     }
347     model_map.erase(model_name);
348     total_memory -= working_memory;
349 }
350
351 extern "C" void load_models() {
352     FILE* model_list_file = fopen((MODEL_DIR + "/model_list.txt").
        c_str(), "r");
353     char model_name[100];
354     std::vector<std::string> models;
355     clear_models();
356     while (fscanf(model_list_file, "%s", model_name) != EOF) {
357         models.push_back(model_name);
358     }
359     for (int i = 0; i < models.size() && total_memory < MEM_LIMIT; i
        ++) {
360         load_model(models[i]);
361         model_list.push_back(models[i]);
362     }
363 }
364
365 extern "C" void init(const char* dir) {
366     if (!is_init) {
367         MODEL_DIR = dir;
368         load_col_type();
369         is_init = true;
370     }
371 }
372
373 void queryRange(Node* root, BOUND_T& bound, FLOAT_T* sum, double&
    count) {
```

```
374 #if 0
375     printf("queryRange\n");
376 #endif
377     memset(sum, 0, sizeof(FLOAT_T) * DATA_DIM);
378     count = 0;
379     _queryRange(root, bound, sum, count);
380 }
381
382 // according to the predication, extract the bound and col_value for
        query
383 void extract_pred(Predication* pred,
384                   int pred_num,
385                   BOUND_T& bound,
386                   COL_VALUE_T& col_value,
387                   MODE mode = MODE::PERFORMANCE) {
388 #if 0
389     printf("extract_pred\n");
390 #endif
391     for (int i = 0; i < DATA_DIM; i++) {
392         bound[i][0] = 1e9;
393         bound[i][1] = -1e9;
394     }
395     if (mode == MODE::PERFORMANCE) {
396         for (int i = 0; i < pred_num; i++) {
397             if (col_map[pred[i].col] >= 0) {  // continuous
398                 bound[col_map[pred[i].col]][0] = pred[i].lb;
399                 bound[col_map[pred[i].col]][1] = pred[i].ub;
400                 if (split_axis_num <= 2) {  // largest model's split
                        axis num is 3
401                     // model_name += std::to_string(pred[i].col) + "
                        _";
402                     col_value.push_back(std::make_pair(pred[i].col,
                        -1));
403                     split_axises[split_axis_num] = col_map[pred[i].
                        col];
404                     split_axis_num++;
405                 }
406             } else {
407                 col_value.push_back(std::make_pair(pred[i].col, int(
                    pred[i].lb)));
408             }
```

```
409             }
410         } else {
411             int count_c = 0;
412             for (int i = 0; i < 7; i++) {
413                 split_axises[i] = i;
414                 col_value.push_back(std::make_pair(i, -1));
415             }
416             split_axis_num = 7;
417             for (int i = 0; i < pred_num; i++) {
418                 if (col_map[pred[i].col] >= 0) {  // continuous
419                     bound[col_map[pred[i].col]][0] = pred[i].lb;
420                     bound[col_map[pred[i].col]][1] = pred[i].ub;
421                     count_c++;
422                 } else {
423                     col_value.push_back(std::make_pair(pred[i].col, int(
                            pred[i].lb)));
424                 }
425             }
426             if (count_c == 0 && pred_num == 3) {
427                 COL_VALUE_T col_value_tmp(col_value.begin() + 7,
                        col_value.end());
428                 col_value = col_value_tmp;
429                 split_axis_num = 0;
430             }
431         }
432         for (int i = 0; i < DATA_DIM; i++) {
433             if (bound[i][0] == 1e9)
434                 bound[i][0] = -1e9;
435             if (bound[i][1] == -1e9)
436                 bound[i][1] = 1e9;
437         }
438     }
439
440     static Answer* _lastAns;
441
442     void clearans() {
443         if (_lastAns != nullptr) {
444             delete[] _lastAns->group_ans;
445             delete _lastAns;
446             _lastAns = nullptr;
447         }
```

```
448 }
449
450 Answer* aqp_group_query(Predication* pred,
451                         int pred_num,
452                         Operation* ops,
453                         int op_num,
454                         COL_T groupBy_col,
455                         MODE mode = MODE::PERFORMANCE) {
456     clearans();
457
458     BOUND_T bound;
459     COL_VALUE_T col_value;
460
461     FLOAT_T* sum = new FLOAT_T[DATA_DIM];
462     double count = 0;
463     split_axis_num = 0;
464
465     Answer* ans = new Answer();
466     _lastAns = ans;
467
468     extract_pred(pred, pred_num, bound, col_value, mode);
469
470 #if 0
471     printf("query init\n");
472 #endif
473     if (groupBy_col != -1) {
474         int groupBy_col_vectorIndex = -1;
475         bool in_pred = !std::none_of(col_value.begin(), col_value.
                end(),
476                                      [groupBy_col](std::pair<int,
                                         int> p) { return p.first ==
                                         groupBy_col; });
477         if (!in_pred) {
478             ans->size = value_num[groupBy_col] * op_num;
479             col_value.push_back(std::make_pair(groupBy_col, -1));
480         } else {
481             ans->size = op_num;
482         }
483         ans->group_ans = new GroupAnswer[ans->size];
484         std::sort(col_value.begin(), col_value.end());
485         groupBy_col_vectorIndex = std::find_if(col_value.begin(),
```

```
            col_value.end(),
                                                    [groupBy_col](std::
                                                        pair<int, int> p)
                                                        { return p.first
                                                        == groupBy_col; })
                                                        -
                                        col_value.begin();
        int bg = 0, ed = value_num[groupBy_col];
        if (in_pred) {
            bg = col_value[groupBy_col_vectorIndex].second;
            ed = bg + 1;
        }
        for (int i = bg; i < ed; i++) {
            col_value[groupBy_col_vectorIndex].second = i;
            Node* root = get_root(col_value);
            queryRange(root, bound, sum, count);
            for (int j = 0; j < op_num; j++) {
                int idx = in_pred ? j : i * op_num + j;
                ans->group_ans[idx].id = i;
                switch (ops[j].op) {
                    case OP::SUM:
                        ans->group_ans[idx].value = round(sum[ops[j
                            ].col] * 10) / 10;
                        break;
                    case OP::AVG:
                        if (count == 0) {
                            ans->group_ans[idx].value = 1;
                        } else {
                            ans->group_ans[idx].value = sum[ops[j].
                                col] / count;
                        }
                        break;
                    case OP::COUNT:
                        ans->group_ans[idx].value = round(count);
                        break;
                    default:
                        break;
                }
            }
        }
    } else {
```

```
520         ans->size = op_num;
521         ans->group_ans = new GroupAnswer[ans->size];
522         std::sort(col_value.begin(), col_value.end());
523         Node* root = get_root(col_value);
524         queryRange(root, bound, sum, count);
525         for (int j = 0; j < op_num; j++) {
526             ans->group_ans[j].id = -1;
527             switch (ops[j].op) {
528                 case OP::SUM:
529                     ans->group_ans[j].value = round(sum[ops[j].col]
                            * 10) / 10;
530                     break;
531                 case OP::AVG:
532                     if (count == 0) {
533                         ans->group_ans[j].value = 1;
534                     } else {
535                         ans->group_ans[j].value = sum[ops[j].col] /
                                count;
536                     }
537                     break;
538                 case OP::COUNT:
539                     ans->group_ans[j].value = round(count);
540                     break;
541                 default:
542                     break;
543             }
544         }
545     }
546     delete[] sum;
547     return ans;
548 }
549
550 /**** KDTree Test Function ****/
551
552 void printKDTree(Node* u, int depth) {
553     if (u->rchild) {
554         printKDTree(u->lchild, depth + 1);
555     }
556     for (int i = 0; i < depth; i++) {
557         printf("  ");
558     }
```

```
559         printf("[%d %.0lf]\n", u->count, u->sum[0]);
560         if (u->lchild) {
561             printKDTree(u->rchild, depth + 1);
562         }
563     }
564
565     void testKDTree(Node* u, int depth) {
566         if (u->lchild) {
567             testKDTree(u->lchild, depth + 1);
568         }
569         if (u->rchild) {
570             testKDTree(u->rchild, depth + 1);
571         }
572         if (IS_LEAF(u) && depth != 0) {
573             if (u->count != 1) {
574                 printf("error\n");
575             }
576         }
577     }
578
579     extern "C" void loadData(FLOAT_T* _data, int n) {
580         clearData();
581
582         dataset_size = n;
583
584         dataset = new FLOAT_T[n * COL_NUM];
585         for (int i = 0; i < n * COL_NUM; i++) {
586             dataset[i] = _data[i];
587         }
588
589         data = new DATA_T[n];
590         for (int i = 0; i < n; i++) {
591             for (int j = 0; j < DATA_DIM; j++) {
592                 data[i][j] = _data[i * COL_NUM + j];
593             }
594         }
595     }
596
597     void clearData() {
598         if (dataset)
599             delete[] dataset;
```

```
600     if (data)
601         delete[] data;
602 }
603
604 extern "C" void build(INT_T* col, int size, int delta_depth, float
        _build_k) {
605     build_k = _build_k;
606     FILE* model_list_file = fopen((MODEL_DIR + "/model_list.txt").
            c_str(), "a");
607     DATA_T* tmp_data = new DATA_T[dataset_size];
608     int tmp_data_size = dataset_size;
609
610     split_axis_num = 0;
611
612     int discrete_axises[12], discrete_axis_num = 0;
613
614     for (int i = 0; i < size; i++) {
615         int c = col[i];
616         if (IS_CONTINUED(c)) {
617             split_axises[split_axis_num] = c;
618             split_axis_num++;
619         } else {
620             discrete_axises[discrete_axis_num] = c;
621             discrete_axis_num++;
622         }
623     }
624
625     // std::copy(data, data + dataset_size, tmp_data);
626     int* d = new int[dataset_size];
627     for (int i = 0; i < dataset_size; i++)
628         d[i] = i;
629
630     std::sort(d, d + dataset_size, [&](int a, int b) {
631         for (int i = 0; i < discrete_axis_num; i++) {
632             if (dataset[a * COL_NUM + discrete_axises[i]] != dataset
                    [b * COL_NUM + discrete_axises[i]]) {
633                 return dataset[a * COL_NUM + discrete_axises[i]] <
                        dataset[b * COL_NUM + discrete_axises[i]];
634             }
635         }
636         return false;
```

```
637     });
638
639     for (int i = 0; i < dataset_size; i++) {
640         for (int j = 0; j < DATA_DIM; j++) {
641             tmp_data[i][j] = dataset[d[i] * COL_NUM + j];
642         }
643     }
644     COL_VALUE_T col_value;
645     for (int i = 0; i < size; i++) {
646         col_value.push_back(std::make_pair(col[i], -1));
647     }
648     std::string model_name = get_model_name(col_value);
649     std::string model_path = get_model_path(model_name);
650
651     model_file = fopen(model_path.c_str(), "wb");
652
653     int l = 0, r = 0;
654
655     while (l != tmp_data_size) {
656         /* get idx */
657         int idx = 0;
658         int tmp = 1;
659         for (int i = 0; i < discrete_axis_num; i++) {
660             idx = idx + tmp * dataset[d[l] * COL_NUM +
                    discrete_axises[i]];
661             tmp *= value_num[discrete_axises[i]];
662         }
663
664         /* get r */
665         r = l;
666         while (r < tmp_data_size - 1) {
667             for (int i = 0; i < discrete_axis_num; i++) {
668                 if (dataset[d[l] * COL_NUM + discrete_axises[i]] !=
669                     dataset[d[r + 1] * COL_NUM + discrete_axises[i
                            ]]) {
670                     goto get_r_end;
671                 }
672             }
673             r++;
674         }
675     get_r_end:
```

```
676
677             /* build */
678             /*
679             -1:     8.1 GB  5.50 s  104 s   9.9e-11
680             -2:     5.0 GB  3.46 s  82.4 s  1.07e-10
681             -3:     3.0 GB  2.10 s  70.1 s  1.17e-10
682             -4:     1.8 GB  1.32 s  62.4 s  2e-10
683             -5:     1.1 GB  0.84 s  57.1 s  6e-10
684             -6:     675 MB  0.57 s  54.0 s  5e-9
685             -6(Of)          0.60 s  52.7 s  5e-9
686             -8:     347 MB  0.34 s  50.2 s  1e-8
687             -10:    263 MB  0.28 s  47.2 s  1e-7
688             -12:    244 MB  0.26 s  45.5 s  5e-6
689             -15:    240 MB  0.27 s  42.3 s  3e-5
690             */
691             max_depth = std::max(1, int(log2(r - l + 1) + delta_depth));
692             // printf("%s %d %d\n", model_name.c_str(), l, r);
693             Node* root = buildKDTree(tmp_data, l, r, 0);
694             saveIDKDTree(root, idx);
695             clearKDTree(root);
696             l = r + 1;
697         }
698
699         fprintf(model_list_file, "%s\n", model_name.c_str());
700
701 #ifdef INFO
702         printf("model_name=%s\nmodel_path=%s\n", model_name.c_str(),
                model_path.c_str());
703         printKDTree(root, 0);
704 #endif
705
706         fclose(model_file);
707         fclose(model_list_file);
708         delete[] tmp_data;
709         delete[] d;
710 }
711
712 void clear_models() {
713         for (auto& model_name : model_list) {
714             clear_model(model_name);
715         }
```

```
716     model_list.clear();
717 }
718
719 extern "C" void clear() {
720     clearData();
721     clearans();
722     clear_models();
723 }
724
725 extern "C" Answer* aqpQuery(Operation* ops,
726                             int ops_size,
727                             Predication* preds,
728                             int preds_size,
729                             COL_T groupBy_col,
730                             MODE mode) {
731     return aqp_group_query(preds, preds_size, ops, ops_size,
        groupBy_col, mode);
732 }
```