

数据库课程设计实验报告

一、小组成员

任务一：杨加佳 15331354
任务二：杨志华 15331370
任务三：杨涵 15331351
任务四：杨金辉 15331357 杨玉楠 15331368

二、实验环境

Linux, Windows, C++

三、实验内容

实现 Ball-Tree 的 C++外存版

任务 1：实现 ball-tree 的建树过程

设计数据结构，按照论文的伪代码实现即可。 N_0 设为 20。

任务 2：实现将 ball-tree 写进外存的功能

按定长记录存储

要求按二进制的页格式存储。缓存页的大小设为 $B = 64K$ 。

每个槽存储一个树节点。

给每个树节点指定 ID，按树节点 ID 查找和存储。

叶子节点和非叶子节点分开存储。（仅供参考）

数据对象直接存放在叶子节点中。（仅供参考）

任务 3：实现从外存中载入 ball-tree 的功能

不要将整棵树载进内存。

用尽可能少的内存消耗，完成尽快的查询

任务 4: 实现查询阶段找到最大内积对象并剪枝的功能
深度优先搜索，按照论文的伪代码实现即可

四、实验思路和设计

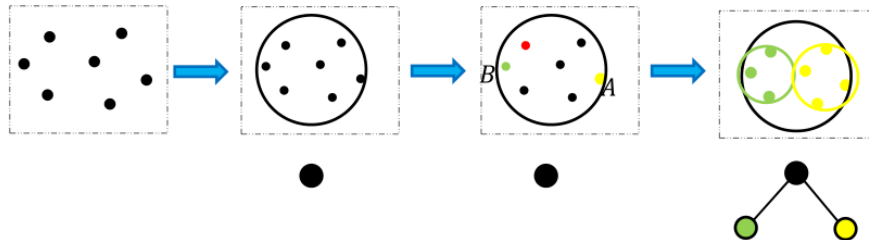
任务 1:

根据 TA 课上给出的建树的说明，其实建树的算法并不难理解

■ Pre-processing

□ MakeBallTree

- Cover the dataset with a ball $B(C, R)$.
- If $|D| \leq N_0$, stop splitting.
- Else find (A, B) using MakeBallTreeSplit.
- Partition the dataset into two parts according to their distance to A and B .



但在具体实现时，遇到的理解上的难题是，如何将数据抽象为一个点。
刚开始我们都以为数据是一个一个的，单个数据该如何抽象为一个点呢？我开始想的是以二维数组的行列下标作为点的横纵坐标，但又发现这样无法表示出点与点之间的距离关系。后来我向其他组请教，才发现我们应该将一组数据抽象为一个点，然后再以点（即每一组数据）为单位进行后续的操作。
为了让大家理解，我还举了一个例子，以注释的形式放入了代码里。

/*

关于Ball Tree数据结构的说明

首先，主体的数据为一个二维数组，我们知道，二维数组其实是由一维数组组成的，那么这个二维数组可以想象成一个表
即每行（一个一维数组）代表一组数据，也就是一个点，行数即为点的数量
每列代表该组数据分量的具体值，列数即为点的维度，应除去第一列，因为第一列为主键id，不能作为计算距离的依据

举个例子，现在有一张表如下：

姓名	年龄	电话	工号
张三	23	123	1
李四	20	456	2
王五	18	789	3

那么用3, 4, 5分别代表他们的姓名, 则这张表可以抽象为如下的二维数组

```
3  23  123  1
4  20  456  2
5  18  789  3
```

其中每行表示一个点, 每列的值相当于该点在不同维度的分量

其实可以看成3个一维数组: (维度为3)

num[1] = {3, 23, 123, 1} 对应点(23, 123, 1)

num[2] = {4, 20, 456, 2} 对应点(20, 456, 2)

num[3] = {5, 18, 789, 3} 对应点(18, 789, 3)

这样就实现了把一组数据抽象为一个点, 然后就能进行距离的判断, 进而进行后面最大内积的计算与求解

理解了 Ball-Tree 的数据结构后, 便要进行节点的设计, 刚开始时, 节点所包含的信息只有 id, 保存数据的二维数组等基础的信息, 但随着整个实验的进行, 为了满足后续任务的需要, 节点的定义在不断修改, 所包含的信息也越来越合理完善, 最终的节点定义如下:

```
struct Node{
    int id;           // 节点的ID, 任务二可能要使用
    Node* left;       // 左节点
    Node* right;      // 右节点
    float** data;      // 如果点的数量小于20才用来存数据, 即相当于这个"圈"里面包含的所有的点的信息
    int count;         // 即这个"圈"里面总共有多少个点, 如果小于20, 则这个点为叶子节点, 不再分裂并进行数据写入, 否则要分裂
    float* center;     // 圆心
    float radius;      // 半径
    int dim;           // 维度
    int DataID;        // 数据的ID, 写入数据页时用
    int LeftID;        // 左节点的ID
    int RightID;       // 右节点的ID
}
```

为了方便, 在节点的设计上就没有分叶子节点和非叶子节点了, 但在后续的处理上我们依然会根据 count 来判断该节点是不是叶子节点, 来进行相应的操作。

具体的实现将在第五个部分说明。

任务 2:

在这部分中, 我们主要要考虑的问题有:

- ① 缓冲页为 64KB 是怎样体现。
- ② 节点跟数据该以怎样的方式存入外存。
- ③ 该用怎样的顺序将节点和数据存入到外存中。

对于问题①, 我们认为缓冲页就是相当于一个存储的数据或节点的 txt 文件, 其最大容量为 64KB。

对于问题②, 我们打算将树节点和数据分开两个不同的部分进行放置, 并且设计出两种不同的存储页:

NodePage:

树节点页中每个槽的大小为 (NodeslotSize): 树节点编号(4 个字节)+左孩子节点编号(4 个字节)+右孩子节点编号(4 个字节)+该节点数据数量(4 个字节)+数据维度(4 个字节)+数据组的编号(4 个字节)+该节点半径+该节点圆心

ID	LeftID	RightID	Count	dim	DataID	Radius	*Center
ID	LeftID	RightID	Count	dim	DataID	Radius	*Center
...							

一个节点页的槽数 (NodeslotNum) = 64KB / NodeslotSize;

页号 (NodePageID) = ID / NodeslotNum;

槽号 (NodeSlotID) = ID % NodeSlotNum;

DataPage:

数据节点页中, 每个槽的大小 (DataslotSize) 为 : 数据组的编号(4 个字节) + 数据组的数据数量(4 个字节)+ 数据组大小

DataID	count	**data
DataID	count	**data
...		

一个数据页的槽数 (DataslotNum) = 64KB / DataslotSize;

页号 (DataPageID) = ID / DataslotNum;

槽号 (DataSlotID) = ID % DataSlotNum;

对于问题③, 我们通过讨论打算用先序遍历的方法对节点进行存储, 而且当出现某个节点的数据数量少于 20 时, 则将其数据存入到相应的数据页中。这样的好处是我们在建树时也是使用先序遍历, 对节点的 ID 命名顺序和数据 ID 的命名顺序也是这样的顺序, 因此我们将节点和数据存入外存时可以直接使用顺序存储, 只需要判断是否一页已满而新开一页, 不需要计算其槽的偏移量。

具体代码实现请参见第五部分。

任务 3:

实现从外存中载入 **ball-tree** 的功能

刚着手这个任务时，我不是很理解从外存中载入 **ball-tree** 具体从代码上要怎么实现，也不知道在外存数据是怎样的一种表现形式，所以同做任务二的队友交流过后才知道他写进外存是通过一条条记录的形式，每条记录代表一个数据，每条记录对应存在一个页表的一个槽中，由于页是固定大小的，所以每一页不可能完整填满，数据页和节点页分开存放。

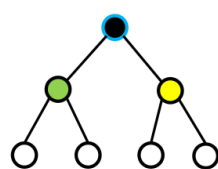
这次我在实现从外存中读入的过程中，对不要将整棵树载进内存的要求感到非常困惑和不解，觉得很抽象不知道怎么在代码中体现出来，但后来看到任务四的内容，是采用 **dfs** 的方法来求最大内积，就想到了可能是在遍历的过程中才逐渐建树，而不是一开始就根据外存中的页将整棵树恢复出来，所以我一开始对 **restore** 函数就理解错了，不是直接载入整棵树，而是只将根节点初始化就行了。

由于任务四在通过 **treesearch** 这个函数来进行 **dfs**，实质上也就是递归，所以我要在它递归前，因为要先求左右儿子，即两个数据集的最大可能内积，所以要先给所在节点赋具体的左右儿子，即根据所给节点的左节点 **ID** 和右节点 **ID** 来算出在哪一页和哪一槽，然后找到相应的记录信息初始化出一个新的节点，分别赋给所给节点的左儿子和右儿子，所以需要有一个函数我在代码文件用 **readNode** 来实现。由于我们数据页和节点页是分开存的，所以如果节点有两种，当节点的数据个数小于 20 时就是存数据的节点，这时候就要从数据页来读取数据赋给节点内的二维数组，这样才能取得节点内的数据进行内积的计算，所以要根据节点的数据 **ID** 来找到数据页，并算出记录的偏移值，读出各个点具体的数据，所以我在代码文件就用 **readData** 这个函数来进行处理，这样就能分别实现数据页和节点页的读取了。

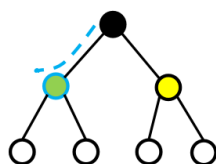
任务 4:

查询阶段找到最大内积对象并剪枝

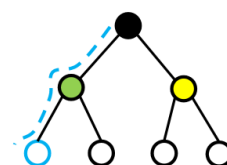
■ Depth First Search



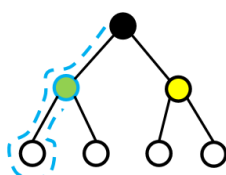
1. Select child
@root



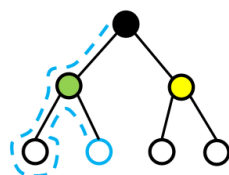
2. Recurse to best
child till 1st leaf



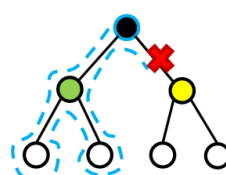
3. Obtain best candidate



4. Try to prune
other children



5. Explore other children
if pruning not possible



6. Save computation
if pruning possible

这里根据要求对树的遍历采用深度优先（DFS）算法，对于一个查询 q ，我们从树的根节点开始，计算该查询和当前节点的最大可能内积 MIP ，并且与已经得到的当前最大内积 $inner$ 进行比较，如果 MIP 比 $inner$ 大，说明该节点之后的数据里存在点使得内积比 $inner$ 大，再进入下一步判断是否为叶节点，如果是叶节点的话则对该节点内的点进行遍历，找到最大内积对象，将内积赋给 $inner$ ，取得其数据 id 。如果不是叶节点，则继续向其子节点遍历，直到找到最大内积对象。如果 MIP 小于等于 $inner$ ，则说明该节点之后之后的叶节点存的数据的内积都不会比 $inner$ 大，最大内积对象不在它之内，就不用继续往下遍历了，返回上一个节点，这就是剪枝。

五、代码结构与实现

任务 1: 具体实现时的大概思路还是按照文献的伪代码来进行的

Algorithm 1 MakeBallTreeSplit(Data S)

```
Pick a random point  $\mathbf{x} \in S$ 
 $A \leftarrow \arg \max_{\mathbf{x}' \in S} \|\mathbf{x} - \mathbf{x}'\|_2^2$ 
 $B \leftarrow \arg \max_{\mathbf{x}' \in S} \|A - \mathbf{x}'\|_2^2$ 
return  $(A, B)$ 
```

Algorithm 2 MakeBallTree(Set of items S)

```
Input – Set  $S$ 
Output – Tree  $T$ 
 $T.S \leftarrow S$  //The set of points in node  $T$ 
 $T.\mu \leftarrow \text{mean}(S)$  //The center of the ball around  $T.S$ 
 $T.R \leftarrow \max_{p \in S} \|p - T.\mu\|_2^2$  //The radius of the ball around  $T.S$ 
if  $|S| \leq N_0$  then
    return  $T$ 
else
     $(A, B) \leftarrow \text{MakeBallTreeSplit}(S)$ 
     $S_l \leftarrow \{p \in S: \|p - A\|_2^2 \leq \|p - B\|_2^2\}; S_r \leftarrow S \setminus S_l$ 
     $T.lc \leftarrow \text{MakeBallTree}(S_l)$  //Left child
     $T.rc \leftarrow \text{MakeBallTree}(S_r)$  //Right child
    return  $T$ 
end if
```

不同的是，我们在 Ball-Tree 的成员变量里添加了 NodeNum 和 DataNum，用来作为节点的 id 及对应的数据的 id，在处理的同时还需要对这两个变量看情况进行递增。

在之前，我们对 id 的命名方式是通过满二叉树父节点和子节点之间的数学关系，直接对子节点的 id 进行赋值的，这种方法是默认二叉树是满的，在存数据的时候会将许多空的节点也写入外存，这个是极其浪费空间的，特别是当数据的数量大了之后。所以用了上面的方法，就能避免某些 id 代表的节点是空的。

任务 2:

写入数据前对各项值初始化:

```
void BallTree::storeInit(const char* index_path) {
    /* 树节点页中每个槽的大小为： 树节点编号(4个字节)+左孩子节点编号(4个字节)+右孩子节点编号(4个字节)+该节点数据数量(4个字节)
    +数据维度(4个字节)+数据组的编号(4个字节)+该节点半径+该节点圆心 */
    NodeslotSize = 4 + 4 + 4 + 4 + 4 + 4 + sizeof(float) + sizeof(float) * root->dim;
    NodeslotNum = PAGE_SIZE / NodeslotSize;
    /* 数据节点页中，每个槽的大小为：数据组的编号(4个字节) + 数据组的数据数量 + 数据组大小
    ps: 因为为定长记录所以数据组大小固定为20个数据* 各每个数据大小，而读时只需读实际数据数量*/
    DataslotSize = 4 + 4 + sizeof(float) * (root->dim + 1) * 20;
    DataslotNum = PAGE_SIZE / DataslotSize;
}
```

对节点进行存储:

```
void BallTree::storeNode(Node* node, ofstream &out, const char* index_path) {
    int NodePageID = node->id / NodeslotNum;
    int NodeSlotID = node->id % NodeslotNum;
    if (NodePageID != CurPageID) {
        CurPageID = NodePageID;
        out.close();
        char *node_path = new char[256];
        sprintf(node_path, "%s/NodePage%d.txt", index_path, NodePageID);
        out.open(node_path, ios::binary | ios::out);
    }
    //对该节点进行写入
    writeNode(node, out);
    //如果节点的数据量小于20, 则需要将其数据写入数据页
    if (node->count < 20) {
        storeData(node, index_path);
    }
    if (node->left != NULL) {
        storeNode(node->left, out, index_path);
    }
    if (node->right != NULL) {
        storeNode(node->right, out, index_path);
    }
}

void BallTree::writeNode(Node* node, ofstream &out) {
    out.write((char*)&node->id, sizeof(int));
    out.write((char*)&node->LeftID, sizeof(int));
    out.write((char*)&node->RightID, sizeof(int));
    out.write((char*)&node->count, sizeof(int));
    out.write((char*)&node->dim, sizeof(int));
    out.write((char*)&node->DataID, sizeof(int));
    out.write((char*)&node->radius, sizeof(float));

    for (int i = 0; i < node->dim; i++) {
        out.write((char*)&node->center[i], sizeof(float));
    }
}
```

对数据进行存储:

```
void BallTree::storeData(Node* node, const char* index_path) {
    int DataPageID = node->DataID / DataslotNum;
    if (CurDataID != DataPageID) {
        CurDataID = DataPageID;
        out2.close();
        char *data_path = new char[256];
        sprintf(data_path, "%s/DataPage%d.txt", index_path, DataPageID);
        out2.open(data_path, ios::binary | ios::out);
    }
    writeData(node, out2);
}
```



```

void BallTree::writeData(Node* node, ofstream &out) {
    out.write((char*)&node->DataID, sizeof(int));
    out.write((char*)&node->count, sizeof(int));
    for (int i = 0; i < node->count; i++) {
        out.write((char*)node->data[i], sizeof(float)*(node->dim+1));
    }
    float* temp = 0;

    for (int i = node->count; i < 20; i++) {
        for (int j = 0; j < node->dim+1; j++) {
            out.write((char*)&temp, sizeof(float));
        }
    }

    delete []temp;
}

```

任务 3:

实现从外存中载入 ball-tree 的功能

实现从外存中载入 ball-tree 的功能在代码中主要功能体现在四个函数，分别为 restoreInit、restoreTree、readNode 和 readData，restoreInit 主要用来初始化树的根节点，为 dfs 求最大内积做好准备，具体代码结构如下：

先打开第 0 页节点页，并读取节点记录里的信息

```

bool BallTree::restoreTree(const char* index_path) {
    char *node_path = new char[256];
    sprintf(node_path, "%s/NodePage%d.txt", index_path, 0);
    ifstream in(node_path, ios::binary | ios::in);
    CurPageID = 0;
    // 先读取根节点信息
    int id, LeftID, RightID, count, dim, DataID;
    float radius;
    // 对含有的信息逐条读取
    in.read((char*)&id, sizeof(int));
    in.read((char*)&LeftID, sizeof(int));
    in.read((char*)&RightID, sizeof(int));
    in.read((char*)&count, sizeof(int));
    in.read((char*)&dim, sizeof(int));
    in.read((char*)&DataID, sizeof(int));
    in.read((char*)&radius, sizeof(float));
    // 数据集的圆心
    float* center = new float[dim];
    in.read((char*)center, sizeof(float)*dim);
}

```

读取完信息后，就可以构造出根节点，如下图：

```
//构造根节点
root = new Node(id, LeftID, RightID, count, dim, DataID, radius, center);
restoreInit();
in.close();
```

接下来是 restoreInit 函数，主要用来初始化所需要用的槽数等常量

```
void BallTree::restoreInit() {
    /*树节点页中每个槽的大小为：树节点编号(4个字节)+左孩子节点编号(4个字节)+右孩子节点编号(4个字节)+该节点数据数量(4个字节)
    +数据维度(4个字节)+数据组的编号(4个字节)+该节点半径+该节点圆心*/
    NodeslotSize = 4 + 4 + 4 + 4 + 4 + 4 + sizeof(float) + sizeof(float) * root->dim;
    NodeslotNum = PAGE_SIZE / NodeslotSize;
    /*数据节点页中，每个槽的大小为：数据组的编号(4个字节) + 数据组的数据数量 + 数据组大小 ps:因为为定长记录所以数据组大小固定为20*/
    DataslotSize = 4 + 4 + sizeof(float) * (root->dim + 1) * 20;
    DataslotNum = PAGE_SIZE / DataslotSize;
}
```

readNode 函数主要用来读取节点页，读取出传入节点的左右节点的所有信息并初始化其左右儿子，具体代码结构如下（只展示左儿子构造，右儿子类似）：

```
void BallTree::readNode(Node* node, const char* index_path) {
    // 先构造node的左节点
    int NodePageID = node->LeftID / NodeslotNum;
    int NodeSlotID = node->LeftID % NodeslotNum;
    CurPageID = NodePageID;
    char *node_path = new char[256];
    sprintf(node_path, "%s/NodePage%d.txt", index_path, NodePageID);
    ifstream in(node_path, ios::binary | ios::in);

    // 寻找所在的槽做偏移
    char* buffer = new char[NodeslotSize];
    for (int i = 0; i < NodeSlotID; i++) {
        in.read(buffer, NodeslotSize);
    }

    // 读取数据构造节点
    int id, LeftID, RightID, count, dim, DataID;
    float radius;
    in.read((char*)&id, sizeof(int));
    in.read((char*)&LeftID, sizeof(int));
    in.read((char*)&RightID, sizeof(int));
    in.read((char*)&count, sizeof(int));
    in.read((char*)&dim, sizeof(int));
    in.read((char*)&DataID, sizeof(int));
    in.read((char*)&radius, sizeof(float));

    float* center = new float[dim];
    for (int i = 0; i < dim; i++) {
        in.read((char*)&center[i], sizeof(float));
    }
    node->left = new Node(id, LeftID, RightID, count, dim, DataID, radius, center);
}
```

主要就是要根据槽 ID，通过循环写入缓冲来进行偏移，然后在逐个字节读取，这样才能准确读出我们想要的节点信息

最后是 `readData`，主要用来读出对应的数据页的具体的数据点，通过节点页来找到数据页的 ID，然后通过偏移来找到数据存储的地方，取出所有数据并赋给节点的二维数组以便求最大内积，如下图：

```
void BallTree::readData(Node* node, const char* index_path) {
    int DataPageID = node->DataID / DataslotNum;
    int DataSlotID = node->DataID % DataslotNum;
    char *data_path = new char[256];
    sprintf(data_path, "%s/DataPage%d.txt", index_path, DataPageID);
    ifstream in(data_path, ios::binary | ios::in);

    char* buffer = new char[2*DataslotSize];
    for (int i = 0; i < DataSlotID; i++) {
        in.read(buffer, DataslotSize);
    }

    int DataID, count;
    in.read((char*)&DataID, sizeof(int));
    in.read((char*)&count, sizeof(int));

    node->data = new float*[node->count];
    float temp;
    for (int i = 0; i < node->count; i++) {
        node->data[i] = new float[node->dim+1];
        for (int j = 0; j < node->dim+1; j++) {
            in.read((char*)&node->data[i][j], sizeof(float));
            temp = node->data[i][j];
        }
    }
    in.close();
}
```

任务 4:

```

bool isLeaf(Node* T); //判断当前节点是否是叶节点

float MIP(Query* q, Node* T); //当前节点可能最大内积

float innerPro(Query* q, float* p, int dim); //计算内积

float innerPro2(Query* q, float* p, int dim);

void TreeSearch(Query* q, Node* T);

int mipSearch(int d, float* query); //d为维度

void LinearSearch(Query* q, Node* T); //当节点为叶节点的时候进行遍历

```

任务 4 所需要用到的函数如上图，其中我们在 test 中用到的是 mipSearch

Algorithm 3 LinearSearch(Query q , Reference Set S)

```

for each  $p \in S$  do
  if  $\langle q, p \rangle > q.\lambda$  then
     $q.bm \leftarrow p$ 
     $q.\lambda \leftarrow \langle q, p \rangle$ 
  end if
end for

```

Algorithm 4 TreeSearch(Query q , Tree Node T)

```

if  $q.\lambda < \text{MIP}(q, T)$  then
  if isLeaf( $T$ ) then
    LinearSearch( $q, T.S$ )
  else
     $I_l \leftarrow \text{MIP}(q, T.lc); I_r \leftarrow \text{MIP}(q, T.rc);$ 
    if  $I_l \leq I_r$  then
      TreeSearch( $q, T.rc$ ); TreeSearch( $q, T.lc$ );
    else
      TreeSearch( $q, T.lc$ ); TreeSearch( $q, T.rc$ );
    end if
  end if
end if

```

而根据文献中给出的伪代码如上图，该部分主要函数就是 LineraSearch 和 TreeSearch，实际具体实现基本上和伪代码相同，除了在 TreeSearch 中，由于我们需要读取数据，所以在 isLeaf(T) 为真时，需要加上

`readData(T, "Netflix/index");` 来读取数据信息，在 isLeaf(T) 为假时，需要加上

`readNode(T, "Netflix/index");` 来读取节点信息。

isLeaf(T): 判断其中的 count 值是否小于 20，是则返回 true，否则返回 false.

MIP(Query* q, Node* T): return $\text{MIP}(q, T) = \langle q, T.\mu \rangle + T.R \|q\|$. (其中, μ 为圆心, R 为半径, $\|q\|$ 为查询的模)

innerPro(Query* q, float* p, int dim): 返回 q 和 p 的内积 $a \cdot b = a_1b_1 + a_2b_2 + \dots + a_nb_n$

innerPro2(Query* q, float* p, int dim): 与 innerPro 基本一样，不同在该函数中 p 的下标从 1 开始，前者从 0 开始

注：这里为了方便 mipSearch 和 TreeSearch 之间变量的转换，我们多设了一个结构体 Query 如下图：

```
struct Query{
    int d;
    float* test;
    int bm;
    float inner;
    Query(int t, float* a) {
        d = t;
        test = new float[t+1];
        test = a;
        bm = 0;
        inner = -9999;
    }
};
```

用于存储查询以及查询维度，当前查询到的最大内积以及最大内积对应的数据的 id。然后再在 mipSearch 中调用 TreeSearch 来查询。

六、实验结果与性能比较

任务 1:

根据 test.cpp 文件进行单纯的建树测试，所需时间为 1.90s

```
Finish reading Netflix/src/dataset.txt
Process exited after 1.902 seconds with return value 0
```

任务 2:

根据 test.cpp 文件进行建树以及写入外存的测试，所需时间为 2.24s

```
Finish reading Netflix/src/dataset.txt
Process exited after 2.244 seconds with return value 0
```

index 文件夹里也成功有文件生成

名称	修改日期	类型	大小
DataPage109	2017/6/7 星期三 ...	文本文档	16 KB
DataPage110	2017/6/7 星期三 ...	文本文档	12 KB
DataPage111	2017/6/7 星期三 ...	文本文档	64 KB
DataPage112	2017/6/7 星期三 ...	文本文档	24 KB
NodePage0	2017/6/7 星期三 ...	文本文档	64 KB
NodePage1	2017/6/7 星期三 ...	文本文档	64 KB
NodePage2	2017/6/7 星期三 ...	文本文档	64 KB
NodePage3	2017/6/7 星期三 ...	文本文档	64 KB
NodePage4	2017/6/7 星期三 ...	文本文档	64 KB
NodePage5	2017/6/7 星期三 ...	文本文档	64 KB
NodePage6	2017/6/7 星期三 ...	文本文档	64 KB
NodePage7	2017/6/7 星期三 ...	文本文档	64 KB
NodePage8	2017/6/7 星期三 ...	文本文档	64 KB
NodePage9	2017/6/7 星期三 ...	文本文档	64 KB
NodePage10	2017/6/7 星期三 ...	文本文档	64 KB
NodePage11	2017/6/7 星期三 ...	文本文档	64 KB
NodePage12	2017/6/7 星期三 ...	文本文档	12 KB

但这里我们可以看到，节点页有 12 页，而数据页有 112 页之多，这里确实是我们的写入算法不够优秀，因为我们为了避免某个叶子节点的数据在写入过程中需要突然需要换页这种复杂情况，是默认每个叶子节点都有 20 组数据的，如果没有就补 0，这样的算法虽然更为简单，能避免一部分比较难 debug 的错误出现，却用了更多的空间，也是各有利弊吧！


任务 3、4：

亦即总体的测试，所需时间为 23.72s

```
Finish reading Netflix/src/dataset.txt
Finish reading Netflix/src/query.txt

Process exited after 23.72 seconds with return value 0
```

dst 文件夹中有 answer.txt 生成

 answer

打开后发现也成功写入了数据（部分数据如下）

1	6797	980	16604
2	6037	981	15205
3	571	982	16954
4	16377	983	5309
5	14240	984	4432
6	6287	985	3962
7	13728	986	12911
8	6287	987	8764
9	15124	988	9340
10	4432	989	15124
11	16377	990	5496
12	15070	991	11064
13	16954	992	17627
14	15582	993	6287
15	5317	994	7624
16	17169	995	16377
17	15582	996	12582
18	6287	997	13081
19	1905	998	7234
20	8644	999	14660
		1000	3962

我们也与另外一个组比对了结果，发现结果是一样的，所以结果应该是正确的。

内存版的运行时间如下：

```
Finish reading Netflix/src/dataset.txt
Finish reading Netflix/src/query.txt
-----
Process exited after 2.761 seconds with return value 0
```

总的来说，无论是建树的时间，还是写入的时间，都比较短，而且由于我们的节点 id 是根据实际的节点数来定的，而不是通过父子节点之间 id 的数学关系来定的，就避免了在按 id 写入时将空节点写入节点页。但同时为了避免在写入某个数据节点的途中换页，我们规定都默认有 20 组数据，不够则补 0，这样虽然能避免一些错误，但是却导致了数据页中有很多地方其实是没有意义的 0，虽然读的时候会避开这些 0，但却占了不少空间。这也是一个遗憾吧，我们在 debug 部分花了太多时间，后面没有时间来优化这一部分了。整个搜索过程下来 20 多秒，虽然和一些比较强的组的 5 秒多相比差一点，但感觉也还行了。

七、心得与体会

杨加佳：

在这次实验中，我主要负责的是建树部分代码的编写。建树这部分代码的算

法其实不难，因为 TA 课上有讲解过，而且文献里也有伪代码，只是刚开始时大家都没能理解怎么把数据抽象成一个点，然后进行后续的操作，所以陷入了一个小瓶颈。我向其他组的同学请教了一番之后，再向组员解释了整体的数据结构以及建树的算法和节点的设计。途中也不断和负责第二部分的同学进行交流，对我们的节点进行了好几轮的完善。

不过当中，我们两人交流修改了一下圆心数组的对应下标，想让它变得更加合理，但当时没意识到圆心在后续算法中的重要作用，也忘记和后面的同学知照一声，结果后面同学使用的还是我们先前的版本，就导致下标出现了问题，他们 3 位同学就这个问题 debug 了一个晚上，让我感到十分抱歉。

最后这几天，大家都来到其中一个同学的宿舍，集思广益，一起 debug，虽然花了很多时间也找不出 bug 时很郁闷，但每个人都出了一份力，最后解决问题的时候真的很开心也很有成就感，这次和大家的合作真的很开心！

杨志华：

本次实验可以说是本学期最难的一个团队 project 了吧？一开始我们组没有先进行小组讨论，而只是简单将工作分配一下，因此在开始时可以说是举步维艰。在前一个组员写完建树部分后，我就着手编写存储部分的代码。在分工时我还没意识到这部分内容的难度，而当我开始编写时却是一脸懵逼。由于本身对文件读写流方面比较弱，而且开始我们没有经过讨论，花费了一整天的时间依然无从下手。后来在与建树部分同学讨论了一波，以及其他小组的帮助下开始大致设计出了两个页。虽然这样的设计方法也许并不是十分理想，但是当发现自己的代码能够顺利将数据存储，还是十分满意的。而这次实验能让我收获的不仅仅是文件流相关的知识，还有我们的团队合作能力，先是小组成员一起 debug 到晚上 2 点多，再是最后关头大家放弃休息时间在最后时刻找出最后的 bug，这都让我感受到了团队工作的氛围以及合作的重要性。

杨涵：

这次负责从外存中读取数据页和节点页来构造树，是对我 `c++` 文件读取知识的一次巨大挑战，以前就不是很喜欢文件输入输出，觉得很麻烦而且很少用，所以也导致我每一次要用到它时都无从下手，对 `c++` 文件读取的细节不是很熟悉，这次由于是采用二进制读写的，一开始用错函数后来才知道二进制文件只能用 `read` 函数来读取，而且在取具体记录时由于不知道怎么转移文件指针，所以后来想到用一个缓冲 `buffer` 来进行槽的偏移，然后到所需记录后再读取相应数据。通过这次实验，大大增强了我对 `c++` 文件 I/O 的编程能力，让我对相关 `api` 有了更深的了解和更熟练的应用，让我对程序内存的控制又有了更深刻的认识，通过页的换入换出来减少内存的消耗，通过将数据写入外存的方式来加快查询速度，对数据库页的存储和记录的更新替换有了更多的理解。

杨玉楠：

这次的实验确实挺难的，我们在阅读论文的基础上来实现整个内容，尽管全英的论文我们也已经翻译过，但是还是需要认真的学习，没有大家的一起努力是不可能在规定时间内完成的，每个人都贡献了自己的力量。每个人也都充分提出过自己的想法，进行过思想的碰撞，才最终能够同心协力完成这次实验。虽然每个人完成的任务都不同，但也都有参与讨论，在小组内不断学习其他人的想法和思维，无形中也是对自己能力的一种提升。在碰到问题的时候，大家能够真正在一起提出自己的想法和意见，一起去解决问题，这样的团队精神也让我很庆幸能在这样的小组里。大家为了这次任务，都付出了很多，也收获了很多，既有新知识的学习，也有旧知识的回顾和巩固。大家都很棒。

杨金辉：

在这次实验当中，我个人最大的体会就是：小组成员之间的交流真的很重要，分工合作，除了分工之外，合作也是不容忽视的，一开始我们小组五个人分配好各自的任务，然后大家完成各自的部分之后进行对接，其中遇到了不少问题，毕

竟因为每个人对实验内容的理解不尽相同，所以在某些地方，不同人有不同的想法，但是我们是在完成同一个项目，除了各个部分的工作做好，将其完美拼接才是最重要的，比如我负责任务四，除了需要任务一建好的树之外，我还需要从文件中读取数据，这就需要和完成任务三的杨涵进行沟通，因为我在查询最大内积对象的时候返回的是该数据的 id，然而在对数据进行计算操作的时候是不需要 id 的，所以如果我没有事先跟他商量好的话，读取的数据就可能读不到 id。然后读数据和写数据之间的衔接也非常重要，稍微有一点错位的话整体就都错了，所以在这上面我们组还是花了不少时间 debug 的。

说到 debug，其实相对比编程，debug 往往要花更多时间，编译器只能帮我们找出语法上的错误，而能否得出正确的结果，只能自己仔细研究，比如我们就采用了最原始的方法，加标签，在有可能出错的地方输出一个标志，看他能否运行到这个地方，以及对有可能出错的数据进行输出，再进行跟踪，看看它是在哪出的问题，再进行修改，这样一点一点的，我们的项目才逐渐成形了，debug 的过程我们也用了两三天，在小组成员的共同努力下，终于得到了正确的结果。

通过这次项目，我对数据库知识的掌握又有了进一步的深入，代码能力和 debug 能力也有了加强，同时在和小组成员的合作之中，从迷茫困惑到一点点成功，收获了喜悦，总的来说是一次很不错的体验。