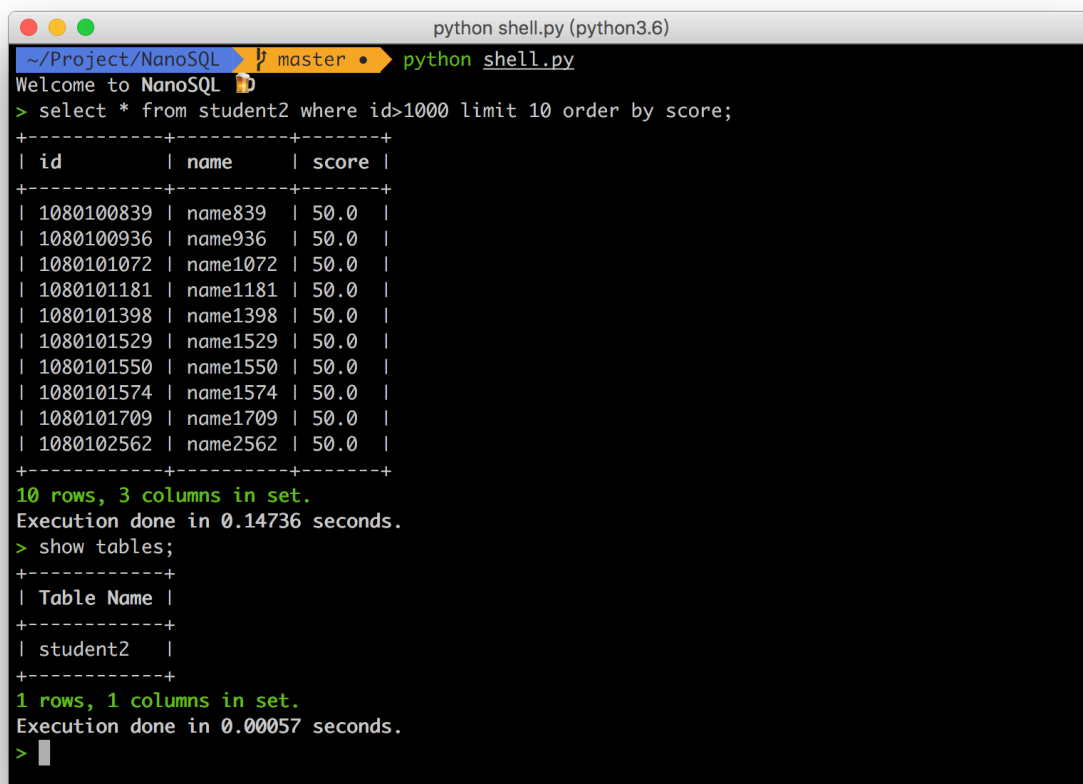


MiniSQL实验报告

3150104894 李海鹏

3150104785 郝广博

项目概况



```
python shell.py (python3.6)
~/.Project/NanoSQL master • python shell.py
Welcome to NanoSQL
> select * from student2 where id>1000 limit 10 order by score;
+-----+-----+-----+
| id      | name   | score |
+-----+-----+-----+
| 1080100839 | name839 | 50.0 |
| 1080100936 | name936 | 50.0 |
| 1080101072 | name1072 | 50.0 |
| 1080101181 | name1181 | 50.0 |
| 1080101398 | name1398 | 50.0 |
| 1080101529 | name1529 | 50.0 |
| 1080101550 | name1550 | 50.0 |
| 1080101574 | name1574 | 50.0 |
| 1080101709 | name1709 | 50.0 |
| 1080102562 | name2562 | 50.0 |
+-----+-----+-----+
10 rows, 3 columns in set.
Execution done in 0.14736 seconds.
> show tables;
+-----+
| Table Name |
+-----+
| student2   |
+-----+
1 rows, 1 columns in set.
Execution done in 0.00057 seconds.
> 
```

NanoSQL 是我们使用Python语言开发的一个简易的数据库系统，支持基本的SQL操作，shell交互，图形界面访问。

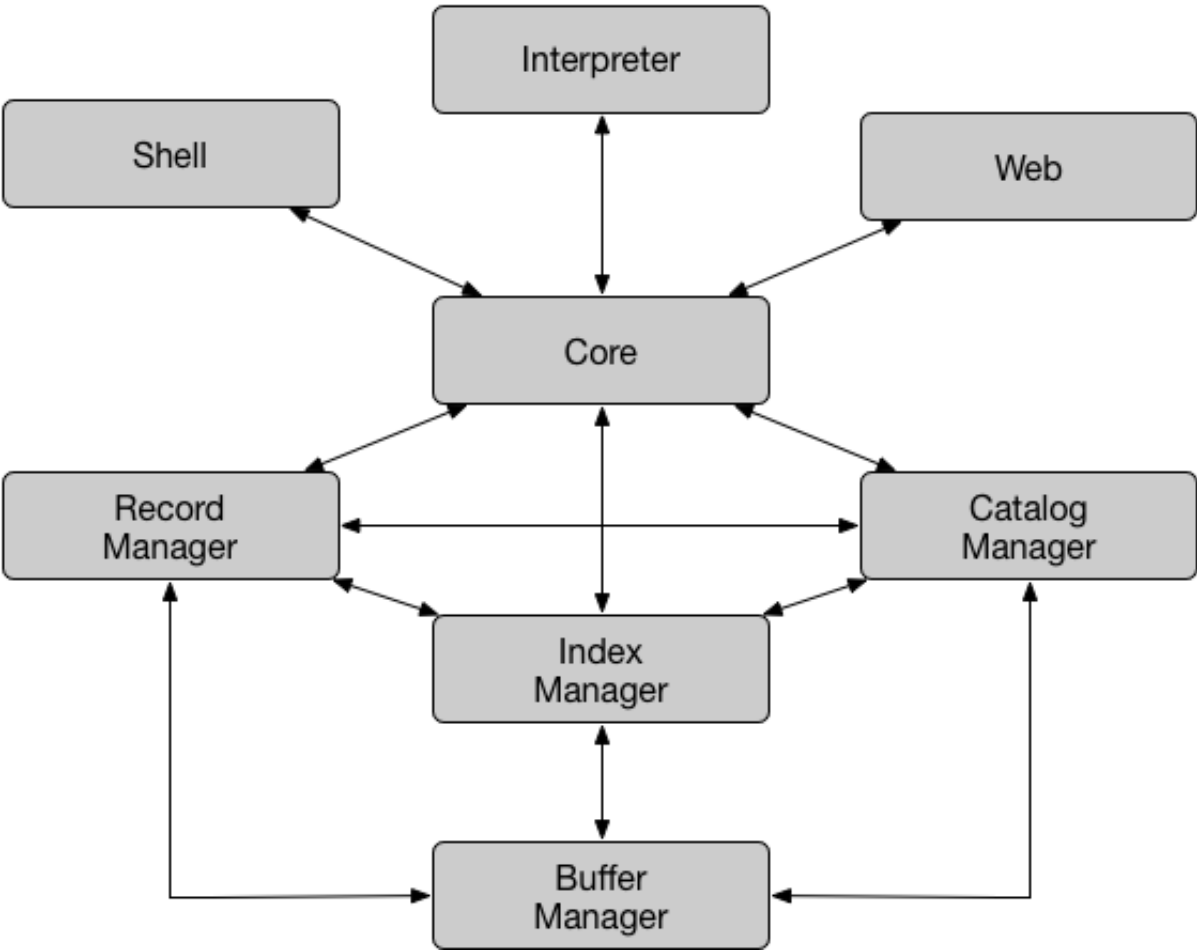
项目地址：[GitHub](#)

分工

模块	负责
shell	郝广博
interpreter	郝广博
core	郝广博
indexManager	李海鹏
BPlusTree	李海鹏
catalogManager	李海鹏
recordManager	李海鹏
bufferManager	郝广博
web	郝广博

架构说明

首先需要说明的是，我们并未完全按照实验指导书提供的模式去设计这个系统，而是自己做了一些架构上的调整。



用户和数据库系统的交互方式有两种，一种是通过shell，另一种是通过web页面，无论是那种方式，交互模块（shell 或者 web）所做的事情就是：把用户输入的SQL语句传送给 core 模块，把 core 返回的SQL执行结果显示给用户。

我们把 interpreter 和 core 模块独立出来了，interpreter 只做SQL语句的解析，通过正则表达式将其解析为字典。在 core 模块拿到SQL语句后，首先通过interpreter做解析，然后再根据解析结果做分发调用。

三大数据处理模块 recordManager、catalogManager、indexManager 除了均和 core 模块、bufferManager 有交互之外，它们之间也有少量的交互。

Shell

总述

shell模块是用来实现和用户的**命令行交互**界面的。

在设计这个模块的过程中，我们不仅要考虑界面的**稳定性**（例如不能因为程序出错而意外结束进程），还要考虑**美观度**（例如我们实现了输出文字的颜色、加粗等样式），以及良好的**交互**（例如易懂的报错信息）。

用户输入数据的读取

通过一个while死循环，实现一直监听用户输入的数据，并且在得到；字符后把本次读取到的字符串合并成一个命令，发送给 core 模块。这里需要说明的是，并非所有的命令都是交由 core 模块处理的，两个特例是：

1. execfile命令
2. quit命令

从文件中读取命令

当用户输入execfile命令时，shell会把用户输入的文件路径提取出来并且打开相应的文件，读取并执行命令。

然而，有时候，会出现一个文件中包含**execfile命令**的情况，为了允许这种情况的发生，我们把 execfile 写成了一个函数，从而利用**递归调用**来实现这个功能。

除此以外，在从一个文件A中调用execfile命令去执行文件B中的命令时，考虑到文件B的路径应该是**相对于文件A所在的路径的**，因此我们会先利用正则表达式提取出文件A的前缀路径和文件B的相对路径做合并，然后再去打开文件B：

```
execFilePath = re.sub(" *;', ", re.sub("^execfile +'", "", command))
execFilePath = re.sub('/[^\s]+'+';', '/', filePath)+execFilePath
execFromFile(execFilePath)
```

查询结果输出

shell模块调用core模块的 execute() 函数，会得到一个**字典**，这个result dict的格式如下：

```
{
  status: 'success' | 'error',
  payload: ...
}
```

因此，根据status和payload的不同，查询结果输出分为三种：

1. 成功文字信息
2. 表格信息
3. 错误文字信息

其中输出表格略微有点繁琐，其他两种情况都很好处理。

输出数据时，使用的是 `print` 函数通过 `end` 参数，来控制是否自动换行。

执行时间的计算

通过Python的标准库 `time`，可以实现对查询执行时间的计算。

只需要在开始调用 `core.execute()` 之前记录开始时间：`timeStart=time.time()`，在查询执行完毕后记录结束时间，二者作差，即可得出查询语句执行的总时间。

需要说明的是，由于 `outputResult()` 是在 `timeElapsed=time.time()-timeStart` 语句之后执行的，因此最终计算出的执行时间是不包括输出数据用时在内的。

辅助函数

为了方便把结果输出到终端，我们设计了一些辅助函数：

- `batchPrint`：将某个字符打印多次
- `printDivider`：打印出分割线
- `bold`：把字符串的最前面和最后面加上加粗标记再返回

Interpreter

总述

`interpreter`模块的主要功能是把输入的SQL语句字符串解析成字典，再返回给`core`模块。

`interpreter`模块的核心函数是 `interpret`，这也是这个模块的入口点，在这个函数中，首先是把 `command` 的最前面的空格通过 `removeFrontSpaces` 函数去除掉，然后再对 `command` 的最前面的字符进行匹配，从而识别出这条SQL语句是属于哪种命令。然后再根据不同的命令类型，去分别调用不同的解析函数，将命令的各项参数解析出来，放到字典的 `data` 字段，和 `operation` 字段合在一起返回给 `core`模块。

最终`interpreter`返回给`core`模块的数据格式如下：

```
{
  operation: 'select' | 'insert' | 'delete' | 'createTable' ...
  data:<Operation Data>
}
```

其中 `operation` 字段是一个字符串，表示SQL语句的类型，`data` 字段是一个Operation Data对象，用来表示SQL语句的具体参数，根据不同的SQL类型，`data` 的格式也会有所变化。列举如下：

Create Table - Operation Data Format

```
{
  tableName,
  fields:[{
    name,
    type: 'char' | 'int' | 'float',
    typeParam:
      <int> //for char type, means max length
      None, //for other types
    unique: <boolean>
  }],
  primaryKey
}
```

Drop Table - Operation Data Format

```
{
  tableName
}
```

Select - Operation Data Format

```
{
  fields: ['field1', 'field2', '*'],
  from: 'table1',
  orderBy: None | 'fieldName', //None means no need to order
  limit: None | 42 //a non-negative, 0 means no need to limit
  where:[
    {
      field: 'field1',
      operand: '=' | '<>' | '<' | '>' | '<=' | '>=',
      value: 'string' | 42
    }
  ] //note: array can be empty, like: []
}
```

Insert - Operation Data Format

```
{
  tableName: 'table1',
  values:[]
}
```

Delete - Operation Data Format

```
{
  from,
  where:[
    {
      field: 'field1',
      operand: '=' | '<>' | '<' | '>' | '<=' | '>=',
      value: 'string' | 42
    }
  ]//same as SELECT
}
```

Create Index - Operation Data Format

```
{
  indexName,
  tableName,
  fieldName
}
```

Drop Index - Operation Data Format

```
{
  indexName
}
```

关于正则表达式

在解析SQL语句的过程中，主要使用的是正则表达式进行各种字符串的处理，如匹配、查找、替换、切分，在代码中，大量出现了 空格* 或者 空格+ 这种语法，主要是用来容许匹配式中出现一个或多个空格，从而能够把一些含有空格但不影响语义的SQL语句正常解析。

Core

总述

core模块是整个NanoSQL系统的**枢纽**，它连接着除了bufferManager以外的所有模块。

core模块的入口点是 `execute` 函数，它以 `command`（SQL语句字符串）为参数，先调用interpreter模块做SQL语句的解析，然后对不同的语句类型做不同的处理，相应去调用底层模块的函数，实现SQL语句所表示的逻辑操作，最后把执行结果返回给shell模块或web模块。

下面列举一下每条语句的**处理逻辑**：

create table

1. 调用catalogManager查看是否已经存在了名字相同的表，如果是的话直接返回报错。
2. 把主键自动设置为unique。（便于下一步处理）
3. 对所有unique的字段，**自动建立索引**。

注：之所以会在create table的时候给所有unique的字段都**自动建立索引**，是为了加速insert速度，在测试样例中，会先进行1万条数据的insert操作，如果此时不是所有的unique字段都被建立过索引的话，就会速度特别特别慢。因为每一次insert操作，都需要遍历整张表，来确保新插入的数据在原来的表里是不存在的。而如果有索引的话，则不需要遍历整张表，而是只需要在B+树上做一次查询即可，经测试，这会产生**极大的速度提升**。

insert

1. 判断表是否存在，如果不存在，直接返回报错。
2. 调用recordManager模块的 insert 函数。

select

1. 判断表是否存在，如果不存在，直接返回报错。
2. 根据catalogManager提供的信息，检查是否每个字段都在表中存在，如果出现**非法字段**，直接返回报错。
3. 把要查询的字段名转成字段的编号。（即，把列的**名字**，转成是**第几列**）
4. 调用recordManager的 select 函数，获取数据。

create index

1. 判断表是否存在，如果不存在，直接返回报错。
2. 把字段名转成字段编号，如果字段不存在，直接返回报错。
3. 分别调用indexManager的 createIndex 函数和catalogManager的 createIndex 函数，建立索引。

drop index

1. 判断表是否存在，如果不存在，直接返回报错。
2. 分别调用indexManager的 dropIndex 函数和catalogManager的 dropIndex 函数，删除索引。

delete

1. 判断表是否存在，如果不存在，直接返回报错。
2. 调用recordManager的 delete 函数。

drop table

1. 判断表是否存在，如果不存在，直接返回报错。
2. 把这张表对应的所有的索引都drop掉。
3. 调用recordManager的 dropTable 函数，删除表对应的记录。
4. 调用catalogManager的 dropTable 函数，删除表的信息。

show tables

1. 调用catalogManager的getTableNames函数，获取到所有表的名字。

2. 根据获取到的数据拼接成可以让shell或者web模块直接渲染显示的二维表格字典。

quit()

quit函数的主要作用是在程序退出时，保存缓存、关闭文件。

因此，程序在退出前，需要先调用quit()函数。

Index Manager

总述

Index Manager负责B+树索引的实现，实现B+树的**创建和删除**(由索引的定义与删除引起)、**等值查找、插入键值、删除键值**等操作，并对外提供相应的接口。

B+树中节点大小应与缓冲区的块大小相同，B+树的叉数由节点大小与索引键大小计算得到。

主要功能接口

createIndex

- 本函数被调用于create index语句之中，其具体作用是**创建一个文件**，将用于保存该索引的数据，同时，本函数还将自动插入该表现存数据的索引内容。
- 传入参数
 - 索引名
 - 表名
 - 列序号
- 返回结果
 - 创建成功与否

dropIndex

- 本函数被调用于drop index语句之中，其具体作用是**删除该表的数据文件**，同时也清空该索引在内存中的数据
- 传入参数
 - 索引名
- 返回结果
 - 删除成功与否

insertIndex

- 本函数被调用于insert values语句之中，其具体作用是在**B+Tree之中插入一条内容**(键，键值)。
- 传入参数
 - 表名
 - 键 (该字段的内容)
 - 键值 (该字段所对应行的行序号)
- 返回结果

- 插入成功与否

select

- 本函数被调用于select语句之中，其具体作用是在该表的索引之中**使用等值查询**，查找符合给定条件的一条内容，随后进行**project**。
- 传入参数
 - 索引名
 - 字段列表
 - 等值查找的待查找值
- 返回结果
 - 仅有一个元素的二维数组(为了与非索引查找的结果结构保持一致)，即为等值查找的结果。

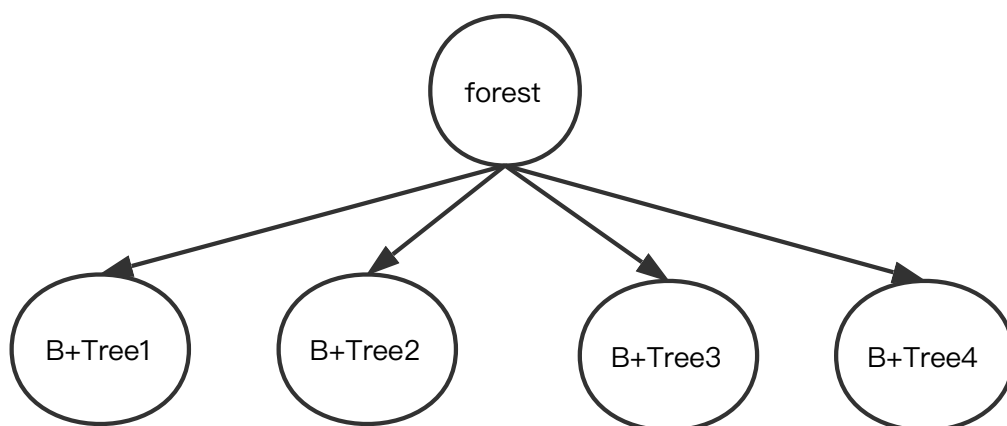
deleteIndex

- 本函数被调用于delete语句之中，其具体作用是删除该索引对应的B+Tree之中符合给定条件的**所有数值及其键值**。
- 传入参数：
 - 索引名
 - 待删除的键的列表
- 返回结果
 - 删除的成功与否

数据结构

forest

此变量为**所有B+Tree的集合**，用python 字典(hash table)的方式存储，查找方式为使用 `indexName` 作为key进行O(1)时间复杂度的查找



B+Tree索引结构请参见B+Tree模块的报告

实现思路及算法

createIndex

- 基本思路为根据所传入的索引名，**新建一个空的文件**，用于保存B+Tree索引数据
- 实现方法：
 - 新建文件：新建空文件的方法为调用buffer manager的 `write`，**写入一个零长度的 b" 内容**，则buffer manager会自动创建一个空文件
 - **导入表中已有数据**：待导入数据的格式为二维数组[(key,value)]获取key的方法为遍历整个已有的表，value的方法可以通过 `recordManager.selectWithNo` 方法获取该key所对应的行序号，然后通过 `sorted` 方法将待插入的数据用key的升序预先排好，之后通过B+Tree的 `bulkload` 方法快速导入至B+Tree

dropIndex

- 基本思路是直接删除所传入的表名所对应的数据存储文件。
- 实现方法：通过调用buffer manager所提供的 `delete` 方法，删除某个文件，并且清除该文件在内存区的buffer内容

insertIndex

- 基本思路是将传入的一行索引数据(key,value)插入至B+Tree
- 实现方法
 - 直接调用B+Tree的 `insert` 方法插入B+TreeNode至B+Tree

select

- 基本思路找到该索引对应的B+Tree，然后通过B+Tree的查找方法快速找到结果所在的行序号，然后通过行序号定位其在原文件之中的所在位置，读取并返回
- 实现方法：
 - 调用B+Tree的 `get` 方法找到该值所在的行序号，然后通过行序号与表大小与块大小的计算，得出该值所在的位置（在文件中的第几个bytes），如果valid，则返回，否则则返回空结果

deleteIndex

- 基本思路是删除B+Tree对应节点的数据，每次调用只能删除一个
- 实现方法是直接调用B+Tree的 `delete` 方法，删除该行对应字段上的内容

B Plus Tree

总述

B Plus Tree实现了封装的B+Tree数据结构，是一个n叉排序树，每个节点有多个孩子，一棵B+树包含根节点、内部节点和叶子节点。根节点可能是一个叶子节点，也可能是一个包含两个或两个以上孩子节点的节点。

本模块为index manager提供了b plus tree数据结构，提供方便的**查询、插入、删除**操作

主要功能接口

bulkload

- 本函数被调用于create index语句之中，其具体作用是为index文件创建一个B+Tree的变量，并且会将传入的数组批量插入至B+Tree之中
- 传入参数
 - 待转化数组
 - order的大小
- 返回结果
 - B+Tree 对象

get

- 本函数被调用于select语句之中，其具体作用为返回B+Tree某个节点(key)所对应的键值(value)
- 传入参数
 - key，即某个表中的某行中的某个字段的值
- 返回结果
 - value，即这一行所在的行序号

insert

- 本函数被调用于insert语句之中，其具体作用为插入某个表中的某行中的某个字段的值和其对应的行序号，以便于以后的查找
- 传入参数
 - key，即某个表中的某行中的某个字段的值
 - value，即这一行所在的行序号
- 返回结果
 - 创建成功与否

delete

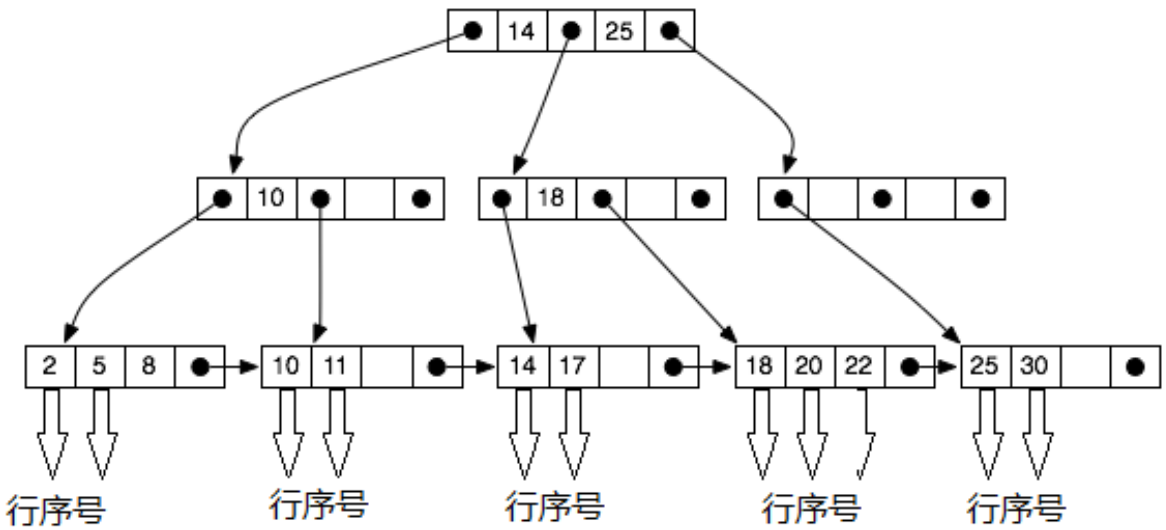
- 本函数被调用于delete语句之中，其具体作用为删除B+Tree中的某个键和其所对应的值
- 传入参数
 - key，即某个表中的某行中的某个字段的值
- 返回结果
 - 删除成功与否

数据结构

B+Tree有以下性质

- 非叶子结点的子树指针与关键字个数相同
 - 非叶子结点的子树指针P[i]，指向关键字值属于[K[i], K[i+1])的子树（B-树是开区间）
 - 为所有叶子结点增加一个链指针
 - 所有关键字都在叶子结点出现
 - 由于限制了除根结点以外的非叶子结点，至少含有M/2个儿子，确保了结点的至少利用率，其查找、插入、删除操作的时间复杂度均为 $O(\log N)$
- 在本模块之中，叶子结点内容为即某个表中的某行中的某个字段的值。叶子结点的链指针指向该

行记录在文件中的位置，具体实现为记录该行的行序号，由于数据为定长存储，根据行序号即可计算得其在文件的具体位置。



- 额外说明：
- 本模块实现的b+Tree为内存中实现，因此在设计的过程中，没有固定每个节点的大小恰好为一个BLOCK的大小。
 - 本模块的B+Tree为纯python实现，相比于同等水平的c++实现代码，更为简单，代码量较小，但是速度有劣势
 - 在本实验的要求的数据量下，经过测试，这些速度劣势完全不可感知。

实现思路及算法

bulkload

给定一组数据记录，我们要在一些关键字段上创建一个B +树索引。一种方法是将每个记录插入一个空树。然而这种做法耗时更长，代价更大，因为每个条目都要求我们从根目录开始，然后到适当的叶子节点进行插入。一个有效的替代方法是使用 **bulkload**

- 按照搜索键按升序对数据条目进行排序
- 我们分配一个空页面作为根，并将一个指针插入到其中的条目的第一页
- 当root已满时，我们拆分根，并创建一个新的根页面
- 将条目插入叶级别上方的最右边的索引页，直到所有条目都被索引为止

get

B +树的根节点表示树的整个范围，每个内部节点都是子间隔。
想要寻找B+树中的值k。从根开始，我们正在寻找可能包含值k的叶子。在每个节点，我们通过比较k与各个元素的大小，找出下一层的节点。内部B+树节点最多 $d \leq b$ 个孩子，其中每个孩子代表不同的子间隔。我们通过搜索节点的键值来选择相应的节点。
实现伪代码如下

```

Function: search (k)
    return tree_search (k, root);
Function: tree_search (k, node)
    if node is a leaf then
        return node;
    switch k do
    case k < k_0
        return tree_search(k, p_0);
    case k_i ≤ k < k_{i+1}
        return tree_search(k, p_{i+1});
    case k_d ≤ k
        return tree_search(k, p_{d+1});

```

insert

m阶B树的插入操作在叶子结点上进行，假设要插入关键值a，找到叶子结点后插入a(查找方法与 `get` 完全相同)，插入情况分以下几种：

- 如果当前结点是**根结点**并且插入后结点关键字数目**小于等于m**，则**算法结束**；
- 如果当前结点是**非根结点**并且插入后结点关键字数目**小于等于m**，则判断若a是新索引值时，转第4点后结束，若a不是新索引值则直接结束；
- 如果插入后关键字数目大于m(阶数)，则结点先**分裂成两个结点X和Y**，并且他们各自所含的关键字个数分别为： $u = \text{大于}(m+1)/2 \text{ 的最小整数}$ ， $v = \text{小于}(m+1)/2 \text{ 的最大整数}$ ；
由于索引值位于结点的最左端或者最右端，不妨假设索引值位于结点最右端，有如下操作：
如果当前分裂成的X和Y结点**原来所属的结点是根结点**，则从X和Y中取出索引的关键字，将这两个关键字组成新的根结点，并且这个根结点指向X和Y，算法结束；
如果当前分裂成的X和Y结点原来所属的结点是**非根结点**，依据假设条件判断，如果a成为Y的新索引值，则转第4点得到Y的双亲结点P，如果a不是Y结点的新索引值，则求出X和Y结点的双亲结点P；然后提取X结点中的新索引值a'，在P中插入关键字a'，从P开始，继续进行插入算法；
- 提取结点原来的索引值b，自顶向下，先判断根是否含有b，是则需要先将b替换为a，然后从根结点开始，记录结点地址P，判断P的孩子是否含有索引值b而不含有索引值a，是则先将孩子结点中的b替换为a，然后将P的孩子的地址赋值给P，继续搜索，直到发现P的孩子中已经含有a值时，停止搜索，返回地址P。

delete

从根开始，找到条目所属的叶L（查找方法与 `get` 完全相同），删除该条目。

- 如果L**至少半满**，完成
- 如果L的条目**数量少于半满**，
 - 如果**同胞**（与L相同的父节点的相邻节点）**超过半满**，则重新分配，借用其中的条目。
 - 否则，**兄弟姐妹完全是半满的**，所以我们可以**合并L和兄弟姐妹**。
 - 如果发生合并，必须从L的父母删除条目（指向L或兄弟）。
 - 合并可以传播到根，降低高度。

Catalog Manager

总述

Catalog Manager 负责管理数据库的所有模式信息，包括：

1. 数据库中所有表的定义信息，包括名称、字段（列）主键在该上索引。
2. 表中每个字段的定义信息，包括类型、是否唯一等。
3. 数据库中所有索引的定义，包括属表、建立在哪个字段上等。

Catalog Manager还必需提供 访问及操作上述信息的接口，Interpreter和API

形象的说，catalogManager是recordManager和indexManager的**指挥员**，后二者的操作总是要先从本模块中获取相应信息之后才可以正常进行。

主要功能接口

createTable

- 本函数被调用于create table语句之中，其具体作用是创建表的定义信息。并保存于 tablesInfo 之中。
- 传入参数：
 - 表名
 - 主键名
 - 字段信息
 - 字段名
 - 字段类型
 - 字段参数(字段长度，仅适用于 char 类型)
 - 是否 unique
- 返回结果
 - 创建成功与否

dropTable

- 本函数被调用于drop table语句之中，其具体作用是在 tablesInfo 变量中删除该表的信息。
- 传入参数：
 - 表名
- 返回结果
 - 删除成功与否

createIndex

- 本函数被调用于create index语句之中，其具体作用是创建某一个表的某一个字段上的index的定义信息。并保存于 indicesInfo 之中。
- 传入参数：
 - 索引名
 - 表名
 - 字段序号
- 返回结果
 - 创建成功与否

dropIndex

- 本函数被调用于drop index语句之中，其具体作用是在 indicesInfo 变量中删除该表的信息。
- 传入参数：
 - 索引名
- 返回结果
 - 删除成功与否

数据结构

tablesInfo

- 本结构存储所有表的定义信息，每次程序启动之时从文件读至内存之中，以python字典的形式存于内存之中
- 每次程序结束之时保存于文件之中，在文件之中以json格式存储
- 结构说明

```
{
  tableName: {
    'primaryKey': ,#主键名称,
    'size': ,#表的大小, 单位为byte
    'fields': [
      'name': ,#字段名
      'type': ,#类型
      'typeParam': ,#'char'类型的长度
      'unique': ,#是否unique
      'index': ,#index的名字(如果有)
    ]
  }
}
```

indicesInfo

- 本结构存储所有索引的定义信息，每次程序启动之时从文件读至内存之中，以python字典的形式存于内存之中
- 每次程序结束之时保存于文件之中，在文件之中以json格式存储
- 结构说明

```
{
  indexName: [
    tableName, #表名
    columnNo #字段序号
  ]
}
```

实现思路及算法

功能类函数

createTable

- 基本思路是将传入的信息保存在 `tablesInfo` 变量之中
- 但是由于我们需要额外附加某些信息，而这些信息没有传入，需要通过计算。我们需要将传入的数据进行扩展，主要包括
 - 表的大小
 - 通过此变量，（由于我们采用的是定长存储策略）我们可以不需遍历表的每一列信息就可以迅速定位到某一行在文件之中的位置以及迅速计算出当前的block是否有足够的剩余空间来存储表中插入的下一列内容
 - 表的大小的计算方法如下（数据的详细存储方式请参见record manager模块的说明）：
 - 首先固定5 bytes的数据，分别保存列序号(no, 4 bytes)和删除标记(validation, 1 byte)
 - `int` 类型为4 bytes
 - `float` 类型为4 bytes
 - `char(n)` 类型为n bytes
 - 表的大小为以上数字之和
 - 通过表的大小来快速定位(时间复杂度为 $O(1)$)某一列在文件中的位置的方法在record manager模块中有详细说明
 - 每一个字段的index信息（即index名）。新表创建时默认设置所有index为 `None` type.
- 扩展之后，我们将以上的数据以 `tablesInfo` 所规定的结构存储，返回成功信息

dropTable

- 基本思路是将某一个表的信息从 `tablesInfo` 内删除
- 通过python `dict.pop()` 函数我们可以方便的删除某一个表的信息，然后返回成功信息

createIndex

- 基本思路是将传入的index信息保存至 `indicesInfo` 变量中
- 同时额外需要将 `tablesInfo` 中的对应表的对应字段的 `index` 信息修改为该index的名称，以完成数据的统一

dropIndex

- 基本思路是将某个index从 `indicesInfo` 变量中删除(通过python `dict.pop()` 函数)
- 同时额外需要将 `tablesInfo` 中的对应表的对应字段的 `index` 信息修改为 `None`，以完成数据的统一

信息查询类函数

以下函数供其他模块(如record manager, index manager, API)调用以获取相应信息

findTable

- 输入表名，作用是返回某一个表的完整信息，包括表的定义，大小，字段的定义，主键，索引等

- 实现方法为python `dict[]` 方法，内部实现采用了hash table的方法
- 如果该表不存在返回None

existTable

- 输入表名，作用是判断某一个表**是否存在**
- 返回值类型为bool type

getTableNames

- 无输入参数，作用是返回**所有表的名称**
- 实现方法为遍历 `tablesInfo`，将所有key存入一个数组，返回该数组

existIndex

- 输入索引名，作用是判断某一个**索引是否存在**
- 返回值类型为bool type

getIndexName

- 输入表名+字段序号，返回**该字段上的索引名**
- 实现方法为通过 `dict[]` 方法查找到该表的信息，再通过字段数组的下标，定位到该字段的信息，查找该字段上的索引名
- 如果该字段上无索引，则返回None

getTableAndColumnNo

- 输入索引名，返回**该索引所在的表名以及字段序号**
- 实现方法为直接通过 `dict[]` 方法找到该index的信息，然后返回所需数据

getFieldsList

- 输入表名，返回该表上的**所有字段信息**
- 实现方法也为 `dict[]`，结果以list的格式返回

getTableSize

- 输入表名，返回该表的一行内容在内存中（也等于在磁盘中）**所占的空间大小**，单位为 bytes
- 实现方法也为 `dict[]`，结果以int type返回

getIndexList

- 输入表名，返回在该表上的**所有索引信息**
- 实现方法为遍历 `indicesInfo`，如果该索引建立在该表上，则保存于list之中
- 返回结果格式为二维数组，其中每一列为[索引名,表名,字段(列)序号]

getFieldNumber

- 输入表名+字段名，返回**该字段在该表之中的序号**
- 实现方法为遍历 `tablesInfo`，查找该字段
- 返回格式为int type

Record Manager

总述

Record Manager负责管理记录表中数据的数据文件。主要功能为实现数据文件的**创建与删除(由表的定义与删除引起)**、记录的**插入、删除与查找操作**，并对外提供相应的接口。

本模块可以实现的查找操作类型有

- 不带条件的查找
- 带条件的查找，包括单条件查找和双条件查找，其中以and连接两个条件。支持的条件类型如下：

- 等值查找(=)
- 不等值查找(<>)
- 区间查找(>, <, <=, >=)

本模块支持的数据类型有

- int， 即32位整数型
- float， 即32位单精度浮点型
- char(n)， 即自定义定长的字符型(其中n>0)

本模块的数据文件保存方式为二进制单文件（每个表单独一个文件），使用buffer manager进行文件的分块单独操作，不支持一条记录的跨块存储。

主要功能接口

createTable

- 本函数被调用于create table语句之中，其具体作用是**创建一个文件**，将用于保存该表的数据
- 传入参数：
 - 表名
- 返回结果
 - 创建成功与否

dropTable

- 本函数被调用于drop table语句之中，其具体作用是**删除该表的数据文件**，同时也清空该表在内存中的数据
- 传入参数：
 - 表名
- 返回结果
 - 删除成功与否

insert

- 本函数被调用于insert values语句之中，其具体作用是在该表之中**插入一行内容**。
- 传入参数：
 - 表名
 - 一行数据内容

- 返回结果
 - 插入成功与否

select

- 本函数被调用于select语句之中，其具体作用是在该表之中查询符合给定条件的所有内容(select)，并且可以选择只返回其中的某些字段内容(project)，此外还可以根据给定的字段进行排序，也可以限制最大输入的行数。
- 传入参数：
 - 表名
 - 选择project哪些字段(默认为全选)
 - 根据哪些条件进行查询(默认为无条件查询)
 - 按哪一列排序(默认为不排序)
 - 限制输出的行数(默认为不限制)
- 返回结果
 - 选择的结果，以二维数组的格式返回，每一行为表中的一行数据

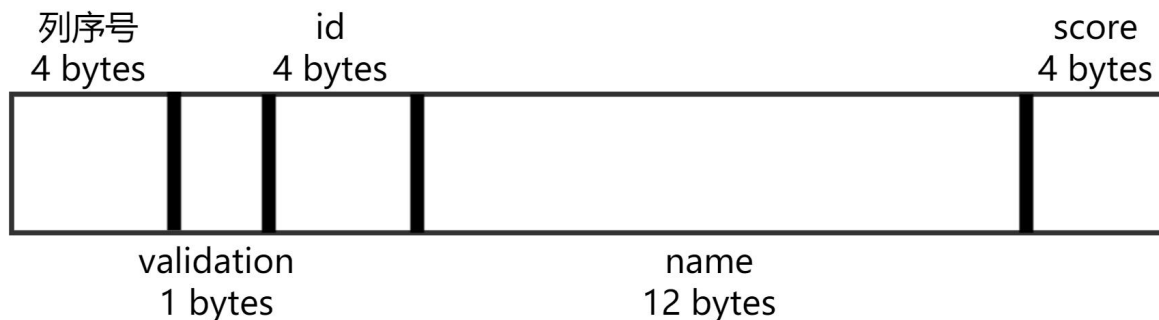
delete

- 本函数被调用于delete语句之中，其具体作用是删除该表之中符合给定条件的所有内容。
- 传入参数：
 - 表名
 - 根据哪些条件进行删除(默认为全部删除)
- 返回结果
 - 删除的行数

数据结构

表内容的存储

- 表内容为定长存储，每一行内容严格按照 `catalogManager` 中给定的表大小存储，行与行之间的内容无分隔符号。
- 以表 `studeng2` 为例



- 首先是附加属性 行序号，表示这一行是全表的第几行。使用32位整数类型存储。
- 接下来是附加属性 `validation`，表示这一行内容是否有效（如果被删除则为无效），使用bool类型存储
- 以下是表的正式内容，第一个为 `id`，使用32位int类型存储(4 bytes)
- 第二个为 `name`，使用12个8位char类型存储(12 bytes)

- 第三个为 `score`，使用32位float单精度浮点类型存储(4 bytes)
- 因此，上述表的每行总大小为 $4+1+4+12+4=25$ bytes，按照buffer manager 的每块大小为4096 bytes计算，每个block可以存放163行记录

实现思路及算法

createTable

- 基本思路为根据所传入的表名，**新建一个空的文件**，用于保存数据
- 实现方法：新建空文件的方法为用过buffer manager，写入一个零长度的 `b''` 内容，则buffer manager会自动创建一个空文件

dropTable

- 基本思路是直接删除所传入的表名所对应的数据存储文件。
- 实现方法：通过调用buffer manager所提供的 `delete` 方法，删除某个文件，并且清除该文件在内存区的buffer内容

insert

- 基本思路是将传入的一行数据**检查重复性**(primary key&unique key)，再内容转化为对应的二进制表示，并且写入在数据文件的末尾
- 实现方法：
 - 检查重复性：调用select语句，查询该primary key, unique key**是否已经存在结果**，如果存在，则报重复错，如果不存在，则继续后续进程
 - 转化为二进制：
 - `行序号`，首先根据数据文件的大小可以直接计算得当前已存在的数据量，以改数字为行号，通过 `struct.pack('i',no)` 则可以将int类型的no转化为4 bytes的二进制表示
 - `validation`，默认为 `b'\x01'`，即有效
 - 对于每一个字段的数据，首先调用catalogManager中的 `getFieldsList` 方法获取该字段的类型，然后调用 `struct.pack('i',intData)`，`struct.pack('i',floatData)` 或 `str.encode(string)` 转化为对应二进制表示
 - 然后**计算存放于文件的位置**，首先根据数据文件的大小可以直接计算得当前已存在的数据量，由于是数据定长存储，可以在 $O(n)$ 时间复杂度下计算获取该写入的block序号，然后**写入到当前内容的尾部**(如果该块剩余空间不足，则填充 `b'\x00'` 至当前块满，然后将真正内容写入到下一块的首部)

select

- 基本思路是**遍历该表的每一列**，对于每一个条件都进行判断，只有**同时满足所有条件才可以加入到最终结果的数组之中**，随后进行**排序操作**，之后进行**project操作**，选择需要的字段，然后返回所得结果的前limit列。
- 实现方法：
 - 如果筛选条件满足**使用索引的条件**(等值查找，有索引)，则调用index manager的查找方法，时间复杂度为 $O(\log N)$ ，否则，则使用遍历查找的方法，复杂度为 $O(N)$
 - **遍历方法**：遍历该文件的每一个block，对于每一个block，按照表大小、字段类型定义进行二进制解码

- 解码方式：
 - 首先根据该条记录的第五个byte(validation)判断，如果为False，则跳过
 - 如果为True，则对后续的二进制内容进行解码,解码方式为 `struct.unpack()` ,可以将二进制内容转化为对应类型的python内置类型
- 判断满足条件的方法：先将条件和内容连接至一个字符串之中，然后调用 `eval()` 对该字符串表示的逻辑表达式进行判断。
- 排序方法：对于筛选后的数组，使用 `sorted(list,key=lambda record: record[orderBy])` 对给定数组，用数组中的第 `orderBy` 个元素进行升序排序。
- **project**方法：因为之前所得到的数组是包含所有字段的信息的，因此我们新建了一个新数组，对于原数组中的每一个元素，只将所需要的字段重新拷贝到新数组。
- 返回前limit个数据的方法：直接返回新数组的[0:limit]的slice

delete

- 基本思路是先**查询**，然后对查询到的结果，将其 `validation` 信息改为**not valid**,并且对删除的数量进行计数
- 实现方法：
 - 查询方法与 `select` 完全相同，此处不再赘述
 - 删除方法为对每一个符合删除条件的内容，将其数据文件中的第5个byte `validation` 改为False(二进制为 `b'\x00'`)。并且对计数器+1

Buffer Manager

总述

bufferManager模块不仅仅是实现了缓存功能，更重要是把对文件的IO操作封装在了一个模块内，使得其他模块不需要自己去处理IO操作，而只需调用bufferManager模块的 `read` 、 `write` 、 `delete` 函数。bufferManager会自动管理缓存，对于其他模块来说，不需要过多的处理缓存问题。

常量定义

`BLOCK_SIZE`：表示一个block有多少个字节，默认是4096（4K）

`MAX_BUFFER_AMOUNT`：表示一个文件对应的buffer最多有多少个block，默认是256

fileList

fileList是一个字典，用来记录所有打开了的文件的信息，下面是一个fileList的样例：

```
{
  'db/student.db': <FileObject>,
  'db/index_name.db': <FileObject>,
  ...
}
```

之所以采用了**字典**来存储这些信息，而不是使用**数组**（Python中也称列表），是因为我们需要频繁的做“根据**file**的名字获取到对应的**fileObject**”这种操作，如果使用数组，需要遍历整个数组才能检索到想要的fileObject，这是十分低效的，而Python的字典本质上就是一张**哈希表**，做这种**键值查询**是非常快的。

一个文件，在打开后，是默认不会自动关闭的，而是直到**程序退出**的时候才统一进行关闭，这是为了避免频繁的文件open/close操作。

bufferList

bufferList用来存储和组织缓存。它是一个**两级的字典**，一级的键是 `filePath`，表示文件名（完整的文件路径），二级的键是 `blockPosition`，表示的是这个文件的哪个block。

也就是说，根据 `filePath` 和 `blockPosition`，可以获取到一个对应的buffer实例，它有 `consistent`、`pinned` 和 `data` 三个键，`data` 的类型是 `bytes`，存放的是block的**实际内容**，`consistent` 的类型是 `bool`，用来表示buffer中的这个block的数据和硬盘上对应的数据是否是一致的（也可以理解为用来表示buffer的数据是否**dirty**），`pinned` 的类型也是 `bool`，用来表示这块buffer是否是被锁定的。

下面是一个bufferList的样例：

```
{
  'db/student.db': {
    12: {
      'data': b'xxxxxxxx',
      'consistent': True,
      'pinned': False
    },
    39: {
      'data': b'xxxxxxxx',
      'consistent': False,
      'pinned': False
    },
    ...
  },
  'db/index_name.db': {
    ...
  },
  ...
}
```

如果想获取名为 `f` 的文件的位置为 `p` 对应的buffer，只需 `bufferList[f][p]` 即可。

如果文件 `f` 在 `bufferList` 中，但是 `p` 并不在 `bufferList[f]` 中，那么调用 `bufferList[f][p]` 会得到 `None`，但是，如果文件 `f` 并不存在 `bufferList` 中，则会出现错误，因为 `bufferList[f]` 已经是 `None` 了，此时 `bufferList[f][p]` 就成了 `None[p]`。

因此，为了保险起见，在获取buffer的时候，需要先**检查f和p是否在bufferList中**：

```
if (filePath in bufferList) and (blockPosition in bufferList[filePath]):  
    return bufferList[filePath][blockPosition]['data']  
else:  
    # handle error
```

openFile

函数openFile的作用是，打开文件，并且把它加入到fileList中。

```
def openFile(filePath):  
    f=open(filePath,'ab')  
    f.close()  
    f=open(filePath,'rb+')  
    fileList[filePath]=f  
    return f
```

之所以先用 ab 模式打开了文件又关闭掉，是因为在用 rb+ 方式打开时，如果文件不存在，并不会自动创建，而是会报错。经过一段时间的研究，我们发现，上面的解决方法已经是相对比较优雅的做法了。

getFile

参数： filePath

返回file的对象，如果 filePath 在 fileList 中不存在对应的文件对象，则自动打开这个文件。

closeAllFiles

关闭所有的文件

read

参数： filePath blockPosition cache=False

返回指定文件的指定block的数据（bytes类型）。

是否需要缓存由 cache 参数控制，默认是 False 。

如果buffer的数量达到了预设的上限，则会先调用 freeBuffer ，释放一个buffer：

```
if (blockPosition not in bufferList[filePath]) and (len(bufferList[filePath])>=MAX_BUFFER_AMOUNT):  
    freeBuffer(filePath)
```

write

参数： filePath blockPosition data cache=False

把 data 的内容写入到指定文件的指定位置或者对应的buffer中（取决 cache 参数）。

如果启用了缓存，则还需要同时把buffer的 consistent 设置为 False 。

delete

参数： `filePath`

删除文件，同时删除该文件对应的所有缓存。

pin

参数： `filePath` `blockPosition`

把指定的buffer标记为 `pinned` 。

blockCount

参数： `filePath`

返回文件的大小（以block记）

freeBuffer

参数： `filePath`

释放一个buffer，采用的替换策略是**Random Replacement**。

相比于LUR和MUR，RR的优势不仅仅在于实现简单，更重要的是，它并不会在buffer的read和write时产生**维护开销**，这使得read和write可以更快速的进行。

我们在设计这个系统时发现，由于buffer开的比较大，实际运行中很少会需要做buffer的释放操作，因此由**加速大概率事件**原理，采用RR替换策略，可以完全去除掉buffer访问频度的统计开销，从而实现更快的IO操作。

save

参数： `filePath`

将该文件对应的所有buffer中， `consistent` 为 `False` 的那些，写回硬盘的文件中。

```
for position in bufferList[filePath]:
    blockBuffer=bufferList[filePath][position]
    if not blockBuffer['consistent']:
        f.seek(int(position)*BLOCK_SIZE,io.SEEK_SET)
        f.write(blockBuffer['data'])
```

saveAll

对 `bufferList` 中的所有文件，依次调用 `save` 函数。

Web

NanoSQL

Please type your SQL here...

Run

Execution result for: `select * from student2 where score=76.5;`

id	name	score
1080100050	name50	76.5
1080100068	name68	76.5
1080100117	name117	76.5
1080100228	name228	76.5
1080100322	name322	76.5
1080100469	name469	76.5
1080100513	name513	76.5
1080100556	name556	76.5
1080100577	name577	76.5
1080100651	name651	76.5

NanoSQL

Please type your SQL here...

Run

Execution result for: `select * from student4;`

ERROR Table student4 does not exist

这个模块是利用[flask](#)框架，搭建了一个非常简易的web服务器，从而给用户提供了图形化的交互界面。

由于这些知识（如http请求的处理、路由配置、模板渲染等等）并不属于数据库系统的范畴，所以这里不再阐述。

在web模块中，并没有做任何SQL语句的处理，而是转交给core模块去做处理。web模块只负责处理和界面相关的逻辑。

个人感悟

郝广博：

虽然NanoSQL远不能达到工业应用的标准，但麻雀虽小五脏俱全，我们在设计和制作它的过程中，对数据库系统的各个模块是如何进行配合、交互的有了更深刻的理解。

李海鹏：

本次project，是我第一次参与的严格意义上的团队项目，对团队合作开发有了一定的心得，期间也与其他组的同学有过交流，感受到人少不一定是坏事，人多不一定是好事。因为人数的增加，会导致组员之间交流成本的急剧增加，经常会导致信息不对称的后果，主要体现在模块与模块之间的对接难度之中。

我们使用了github代码仓库，代码改动可以随时互相更新；小组总共两位成员，对数据结构和模块之间的接口的讨论成本较低；并且有撰写说明文档的好习惯，使得我们可以将主要精力放在程序本身的代码实现之上，而没有出现其他很多小组出现的拼接各个模块困难的问题。

在数据库的原理方面，由于我负责的是catalog, record, index manager三个模块，十分贴近课本所学的理论知识。在模块的实现之中，深入理解了数据库底层的文件保存方式，掌握了B+Tree的原理以及使用方法，认识到python作为动态语言的优越性以及性能的不足性，并且在自己完成的NanoSQL与现有的MySQL等软件的对比之中，深切的体会到真正的SQL的复杂性，和数据库这门学科的博大精深。