MiniSQL

一、 实验目的

1. 设计并实现一个精简型单用户 SQL 引擎(DBMS)MiniSQL，允许用户通过字符界面输入 SQL 语句实现一些简单的数据库管理操作。
2. 通过对 MiniSQL 的设计与实现，提高学生的系统编程能力，加深对数据库系统原理的理解。

二、 系统需求

1. 数据类型：支持int，float，char(n)
2. 表定义：一个表最多定义32个属性，各属性可以指定是否为unique；支持unique属性的主键定义
3. 索引的建立和删除：对于表的主键自动建立B+树索引，对于声明为unique的属性可以由用户指定建立或删除索引
4. 查找记录：可以通过指定用and连接的多个条件进行查询，支持等值查询和区间查询
5. 插入和删除记录：支持每次一条记录的插入；支持每次一条或多条记录的删除
6. 语句格式与MySQL一致

三、 实验环境

1. Python 3.8.1

四、 模块设计

以下只阐述本次实验中由本人完成的部分，完整全部内容请查看小组的实验报告。

4.1 API

该模块负责**衔接顶层的interpreter以及底层的各个模块**。本次实验中API由本人完成的部分有创建表、删除表、创建索引、删除索引以及插入记录。采用面向对象编程的思想，将API封装为一个类，其中由我完成的部分接口如下：

```
class API():
    def __init__(self, catalog, buffer, record, index)
    def create_table(self, tbl_name, tbl_pky, tbl_attributes)
    def drop_table(self, tbl_name)
    def create_index(self, idx_name, idx_tbl, idx_key)
    def drop_index(self, index_name)
    def insert_record(self, table, value, attr=None, import_flag = False)
```

其中，每个函数都没有返回值。这是因为结果的打印都直接在API内进行，不需要再经过上层的模块。我们在设计时遵循的一个原则是，**先检查后实现**：需要写一个文件时，需要首先检查写入是否合法，不合法会**抛出异常**以阻止后续的写入，只有所有检查都通过，我们才会写入一个文件。

初始化函数`__init__`读入底层的模块，以此来保证各个对象的唯一性，同时方便各个方法调用底层的函数。

创建表函数`create_table`读入表的名字、表的主键以及表的属性，需要实现：在catalog中加入该表、为该表建立主键索引、创建相应的文件。

删除表函数`drop_table`读入表的名字，需要实现：在catalog中删除该表、删除建立在该表上的索引、删除相应的文件。

创建索引函数`create_index`读入索引的名字、表的名字以及键的名字，需要实现：在catalog中加入该索引、建立B+树索引、将其保存到文件中。

删除索引函数`drop_index`读入索引的名字，需要实现：在catalog中删除该表、删除相应的文件。

插入记录函数`insert_record`读入表名、各属性的值，指定的属性名以及一个是否为批量插入的指示，需要实现：插入该条记录到相应的表中、维护所有建于该表上的索引。

本人实现的部分中没有特别的数据结构，主要是列表以及字典，略去不谈。

4.2 Catalog Manager

该模块负责**管理表的目录和索引的目录**。为了方便管理，首先设计一个表示列的对象，包括名字、类型等属性，如下：

```
class Attribute():
    def __init__(self, name, type='20s', length=20, uniqueness=False):
        self.name = name
        self.uniqueness = uniqueness
        self.type = type
        self.length = length
```

其次再设计一个表示表的对象，其需要包括表名、主键、列等属性，如下：

```
class Table():
    def __init__(self, table_name, primary_key, number_attributes):
        self.table_name = table_name
        self.primary_key = primary_key
        self.attributes = []
        self.number_attributes = number_attributes
```

而由于索引包含的信息较少，只有索引名、键名和表名，因此不单独为其设计对象。直接采用字典结合列表的方式进行管理。

除去上述结构，剩下的就是主体的`catalog_manager`对象，该对象我设计了很多方法，其中**大多数是检查性质**的，为了方便上层模块的调用。由于数量众多，这些检查的方法并不在此全部列出。以及，考虑到目录文件存储的信息较少，因此我们在设计上，`catalog_manager`是直接管理目录文件的，并不需要经过buffer的管理。该类的部分接口如下：

```
class catalog_manager:
    def __init__(self)
    def save(self)
    def table_exists(self, name)
    def table_not_exists(self, name)
    def key_not_exists(self, tbl_name, key)
    def create_table(self, tbl_name, primary_key, attrlist)
    def create_index(self, index_name, tbl_name, key)
    def check_record_files(self, table)
    ...
```

初始化函数`__init__`需要实现：检查目录文件的完整性、从文件中读取目录以及检查各个表文件和索引文件的完整性。在读入目录信息时，`catalog_manager`维护一个以表名作为键的**字典**以及一个以索引名作为键的**字典**。

保存函数`save`需要实现：将目录写入文件保存起来。

检查函数`table_exists`读入一个表名，需要实现：检查该表名是否已经存在，如果存在则抛出异常。

检查函数`table_not_exists`读入一个表名，需要实现：检查该表名是否已经存在，如果不存在则抛出异常。其他的检查函数大同小异，在此不再赘述。

创建表函数`create_table`读入表名、主键以及各列的属性，需要实现：在catalog中加入该新建的表。创建索引的函数与其类似。

删除表函数`drop_table`读入表名，需要实现：在catalog中删除该表。删除索引的函数与其类似。

检查表文件函数`check_record_files`读入一个表名，需要实现：检查该表的表文件是否存在，如果不存在，则抛出异常。检查索引文件的函数与其类似。

4.3 Index Manager

该模块负责**B+树索引的维护**。首先设计了一个表示节点的类，该类包含的成员有键值的列表、孩子节点的列表、保存到文件中需要用到的块的id（其实就是对B+树做一次level order之后的序号）、该节点是否为叶子结点的指示以及该节点的父母。树的order没有保存在结点中，而是保存在树的对象中，这是为了避免同样的信息被保存多次，造成信息冗余。部分接口如下：

```
class Node():
    def __init__(self, isleaf=False, parent=None):
        self.keys = []
        self.children = []
        self.bid = 0
        self.isleaf = isleaf
        self.parent = parent
    def index_of_insert(self, key)
```

此处唯一展示的方法`index_of_insert`读入一个键值，返回这个键值插入该节点后的下标。其他方法都非常简单，主要是用于判断节点处于何种状态，不叙述。

模块的主体则是名为`index_manager`的类，设计时本人尽量将与除目录外，所有与索引有关的操作都放到这里实现，比如文件的创建、读写与移除。实际编写时，考虑到编程的便捷度，没有写删除键值的方法，维护索引的方法目前只有重建索引。同时，由于直接在相应的块中维护B+树较为困难，存在着诸多如类型转换的问题，B+树索引在内存中建立，保存到文件后，搜索时不会再重建B+树。其主要接口如下：

```
class index_manager():
    def __init__(self, buffer_manager)
    def search(self, name, value, type, length)
    def search_domain(self, name, value, operate, type, length)
    def create_index(self, index_name, addresses, values, order)
    def save_Bplus(self, index_name, type, length)
    def drop_index_file(self, index_name)
```

初始化函数`__init__`读入buffer manager并保存，确保其唯一性，同时要保存B+树索引的order以及根结点。

搜索函数`search`读入索引名、键值、键的类型以及长度，需要实现：在相应的B+树索引文件中查找该键值。如果键值存在则返回其在表文件中的地址指针，否则返回`False`。

区域搜索函数`search_domain`读入索引名、键值、条件、键的类型以及长度，需要实现：在相应的B+树索引文件中查找满足条件的键值。如果存在满足条件的键值则返回它们的指针。（此函数不是由本人实现）

创建索引函数`create_index`读入索引名，order、键值列表及其地址列表，需要实现：建立相应的B+树索引。

保存索引函数`save_Bplus`读入索引名、键值的类型及其地址，需要实现：将相应的B+树索引保存到文件中。

删除索引函数`drop_index_file`读入索引名，需要实现：将该索引文件移除。

五、 模块实现

4.1 API

本模块接收来自interpreter的用户输入，调用底层的模块并输出结果。除了将各个函数集成为一个对象外，没有特殊的数据结构。**本模块的方法，大致流程都是：处理interpreter的输入、检查合法性、调用底层模块。**

初始化函数只有简单的赋值引用，确保各个模块的对象的唯一性。

除了上文提到的公开的方法，还有四个私有的方法需要先说一说。首先是字符串处理函数，将用户输入的字符串转换为合适的类型。如对于值，要将字符串转换为整数或者浮点数；对于类型，要将`int`转换为`i`（因为我们采用`struct`包来读写二进制文件，将类型用`i`、`f`、`xxs`来表示会更为方便）。

```
    def __string_process(self, table, value):
        for i, item in enumerate(self.catalog.tables[table].attributes):
            if item.type[-1] == 's':
                value[i] = value[i][1:-1]
                value[i] = str(value[i])
                if len(value[i]) > item.length:
                    raise Exception("ERROR: Data too long for column
'%s'"%item.name)
                value[i] = value[i].encode('utf-8')
            elif item.type == 'i':
                value[i] = int(value[i])
            elif item.type == 'f':
                value[i] = float(value[i])
        return value
```

条件处理函数，主要用于处理`where`字句中输入的条件。其将一个条件转换为下标+条件代码+值的三元列表，代码如下：

```
    def __condition_process(self, table, conditions):
        conditions = [list(item) for item in conditions]
        for item in conditions:
            if item[1] == '<':
                item[1] = 0
            elif item[1] == '<=':
                item[1] = 1
            elif item[1] == '>':
                item[1] = 2
            elif item[1] == '>=':
                item[1] = 3
            elif item[1] == '=':
                item[1] = 4
            elif item[1] == '<>':
                item[1] = 5
            else:
                raise Exception('SYNTAX Error: There is illegal operator in your
SQL syntax')
            idx = self.catalog.index_in_table(table, item[0])
            if self.catalog.tables[table].attributes[idx].type == 'i':
                item[2] = int(item[2])
            elif self.catalog.tables[table].attributes[idx].type == 'f':
                item[2] = float(item[2])
            else :
                item[2] = item[2][1:-1]
                item[2] = item[2].encode('utf-8')
            item[0] = idx
        return conditions
```

检查唯一性的函数，读入表名与即将要插入的值，检查其是否破坏了该表的的唯一性约束。代码如下：

```
    def __check_uniqueness(self, table, attr, value):
        check_record, check_ptr = self.record.scan_all(table, [], attr)
        for i in range(len(check_record)):
            for j, column in enumerate(self.catalog.tables[table].attributes):
                if column.uniqueness is True:
                    if len(column.type) == 1 and check_record[i][j] == value[j]:
                        raise Exception('INVALID VALUE Error: Duplicate entry '
+ column.name + ' for ' + str(value[j]))
                    elif len(column.type) != 1 and check_record[i]
[j].strip(b'\x00') == value[j]:
                        raise Exception('INVALID VALUE Error: Duplicate entry '
+ column.name + ' for ' + str(value[j]))
```

第四个是更新所有索引的函数，读入一个表名，更新所有建立在该表上的索引。实现很简单，没什么好说的，代码如下：

```
    def __all_index_update(self, table, attr):
        result_value, result_ptr = self.record.scan_all(table, [], attr)
        tmp = []
        for item in self.catalog.indices.keys():
            if self.catalog.indices[item][0] == table:
                idx = self.catalog.index_in_table(table,
self.catalog.indices[item][1])
                tmp.append([item, idx,
self.catalog.tables[table].attributes[idx].type,
self.catalog.tables[table].attributes[idx].length])
        for index_name, i, type, length in tmp:
            key_value = [item[i] for item in result_value]
            order = (4096-2-1-2) // (length + 2) + 1
            self.index.create_index(index_name, result_ptr, key_value, order)
            self.index.save_Bplus(index_name, type, length)
```

以上四个私有化方法，**提高了API中代码的复用性**，也简化了其他公开的方法的编程。

再说上述提到的公开的方法。创建表函数首先将主键的唯一性设为真，然后检查输入值的数量是否与该表的列数相等，再将各输入值从字符串转换为适合的类型。最后先调用catalog的方法，因为表名的合法性等性质会首先在catalog检查，再调用record manger以及index manger的创建函数。伪代码如下：

```
    def create_table(self, tbl_name, tbl_pky, tbl_attributes):
        set the primary key to be unique
        check if the name for each attribute is unique
        check if the number of input values matches the number of columns
        transform the input for the bottom modules
        self.catalog.create_table(tbl_name, tbl_pky, tbl_attributes)
        self.catalog.create_index(tbl_name, tbl_name, tbl_pky)
        self.record.create(tbl_name, tbl_attributes)
        self.index.create_index_file(tbl_name)
```

删除表函数要做的事情相对简单，只需要注意检查删除的表是否存在，以及删除一个表后还要将建立在该表上的所有索引都删除。伪代码如下：

```
    def drop_table(self, tbl_name):
        self.catalog.drop_table(tbl_name)
        tmp_indices = []
        for index in self.catalog.indices.keys():
            if index is based on tbl_name:
                tmp_indices.append(index)
        for index in tmp_indices:
            self.drop_index(index)
        # drop the table at last
        self.record.drop_record_file(tbl_name)
        self.catalog.save()
```

创建索引的函数首先在catalog中创建索引（这一步会检查该索引的创建是否合法），然后处理输入以读取该表中所有的键值及其地址，最后利用该表中所有的键值及其地址建立B+树索引并保存。伪代码如下：

```
    def create_index(self, idx_name, idx_tbl, idx_key):
```

```
        # duplicate, existence and uniqueness is checked in this call
        self.catalog.create_index(idx_name, idx_tbl, idx_key)
        find the index, length and type of idx_key
        # read all the records
        attrlist = []
        for item in self.catalog.tables[idx_tbl].attributes:
            attrlist.append((item.name, item.type, item.length,
item.uniqueness))
        records, addresses = the values and addresses of all the records
        # extract the values
        values = [rec[key_idx] for rec in records]
        # order = (4096-2-1-2) // (length of key + 2) + 1
        order = calculate the order
        self.index.create_index(idx_name, addresses, values, order)
        self.index.save_Bplus(idx_name, type, length)
```

删除索引函数很简单，只需要依次调用各个模块中删除索引的函数。删除操作的合法性也会首先在catalog中检查。伪代码如下：

```
    def drop_index(self, index_name):
        # existence is checked in this call
        self.catalog.drop_index(index_name)
        self.index.drop_index_file(index_name)
        self.catalog.save()
```

插入记录函数，首先要对该插入操作的合法性进行检查，包括检查文件是否存在、表名是否存在等。然后调用前述的私有化方法对输入进行处理、检查。上述两步完成后，才进行数据插入表文件以及维护索引。此处`import_flag`的作用是指示该方法是在处理单条插入还是批量插入。如果是单条插入，则在插入完成后需要更新索引；如果是批量插入，则没有必要每一次插入都更新索引，**只需要在批量插入完成后更新一次索引即可**。这可以加快程序的运行。伪代码如下：

```
    def insert_record(self, table, value, attr=None, import_flag = False):
        self.catalog.table_not_exists(table)
        self.catalog.check_record_files(table)
        self.catalog.check_index_files(indices based on table)
        '''transform the input to the correct format'''
        value = self.__string_process(table, value)
        if attr is None:
            attr = [[item.name, item.type, item.length, item.uniqueness]
                    for item in self.catalog.tables[table].attributes]
        else:
            pass
        '''read a record here and check all the uniqueness'''
        self.__check_uniqueness(table, attr, value)
        attribute = [[item.name, item.type, item.length, item.uniqueness]
                     for item in self.catalog.tables[table].attributes]
        self.record.insert(table, attribute, value)
        if import_flag is False:
            self.__all_index_update(table, attr)
```

由于API模块是最后编写的模块，此时我们已经可以在命令行查看程序的输出以及输入。测试方法基本就是对每一个方法单独在命令行进行测试，也因此我们没有留下单独测试API的代码。

4.2 Catalog Manager

本模块的实现较为简单，且大多的做法都很简单，因此不会对每一个方法都加以解释。因为catalog的信息较少，因此设计上catalog manager可以直接读写文件。而又考虑到二进制文件是更为高效的存储方式，我们利用了`struct`包来帮助我们读写二进制文件。

首先是对象的初始化方法。由于目录的特殊性，初始化的时候其需要打开目录文件（此处先检查文件是否存在，若不存在则会阻止程序运行），读出表目录以及索引目录。重建索引完毕后，还会先检查一次各个文件是否都存在。伪代码如下：

```
    def __init__(self):
        check_catalog()
        self.tables = {}  # dictionary for all tables
        self.indices = {}  # dictionary for all indices
        # build tables
        os.chdir(sys.path[0])
        file = open('./catalog/table_catalog.dat', 'rb')
        table_num = the number of all the tables
        for i in range(table_num):
            read the information and rebuild the catalog for tables
        file.close()
        # build indices
        file = open('./catalog/index_catalog.dat', 'rb')
        index_num = the number of all the indices
        for i in range(index_num):
            read the information and rebuild the catalog for indices
        file.close()
        # check existence
        for table in self.tables.keys():
            self.check_record_files(table)
        for index in self.indices.keys():
            self.check_index_files(index)
```

保存函数的操作很简单，就是简单地打开文件并且把catalog写入文件。写入表目录时，先保存表的数量，再依次保存每个表的名字、主键以及属性；写入索引目录时，先保存索引的数量，再依次保存每个索引的名字、所基于表的名字以及键的名字。伪代码如下：

```
    def save(self):
        # save tables
        os.chdir(sys.path[0])
        file = open('./catalog/table_catalog.dat', 'wb+')
        write the number of tables
        for table in self.tables:
            write the name of the table
            write the name of the primary key
            for attr in self.tables[table].attributes:
                write the information of each attribute
        file.close()
        # save indices
        file = open('./catalog/index_catalog.dat', 'wb+')
        write the number of indices
        for index in self.indices:
            write the name of the index
            write the table name of the index
            write the name of the key
        file.close()
```

检查性质的方法只介绍一个检查表名是否已经存在的方法，实现非常简单，即遍历一次表目录，如果发现重名则报错。代码如下：

```
    def table_exists(self, name):
        if name in self.tables.keys():
            raise Exception("INVALID IDENTIFIER: Table '%s' exists" % name)
```

创建表、删除表、创建索引以及删除索引的流程都是一样的，都是**先检查后创建/删除**。因为实现简单，也仅展示其中一个的代码。代码如下：

```
    def create_index(self, index_name, tbl_name, key):
        self.key_not_exists(tbl_name, key)
        self.key_not_unique(tbl_name, key)
        self.index_exists(index_name)
        self.indices[index_name] = [tbl_name, key]
```

检查文件是否存在的函数也很简单，简单来说就是调用`os`包的方法，不再赘述。

测试部分，主要就是简单地调用接口的函数并多次运行测试代码。主要查看两点：**保存的文件内容是否正确**以及**能否正确地从文件中重建目录**。测试代码如下：

```
if __name__ == "__main__":
    t = catalog_manager()
    print(t.tables)
    print(t.indices)
    print(t.is_index_key('xyz', 'sid'))
    print(t.is_index_key('xyz', 'xxx'))
    t.create_table('xyz', 'sid', [['sid', '11s', 11, True], [
                   'name', '3s', 3, False], ['sex', 'i', 20, False]])
    t.create_table('abc', 'sid', [['sid', '20s', 20, True], [
                   'name', '3s', 3, False], ['sex', 'i', 20, False]])
    t.save()
```

4.3 Index Manager

本模块用到的数据结构就是B+树。因为在课程上已学习过，此处不对B+树做过多介绍。关于结点对象的定义，重要的部分也在前面讲过了，不再赘述。这里着重讲一下B+树的建立、搜索以及保存的过程。

建立索引的函数非常简单，就是调用了两个该模块的方法。代码如下：

```
    def create_index(self, index_name, addresses, values, order):
        self.build_Bplus(index_name, addresses, values, order)
        self.print_tree()
```

其中第二个函数是对B+树进行level order，得到每个结点保存在文件中时的block id。第一个方法就是建立B+树的方法，其本质就是初始化根结点、order等信息后，不断地向其中插入数据。因此直接看插入函数。第一步是搜索该值插入B+树后应处于的叶子结点，将其保存在`leaf`中。第二步是向`leaf`中插入相应的键值以及其地址。第三步则是检查B+树的性质有没有被违背，如果有，则需要进行split等操作来维护其性质。伪代码如下：

```
    def __insert(self, index_name, address, value, order):
        root = self.root
        leaf = self.__search_leaf(root, value)

        index = leaf.index_of_insert(value)
        leaf.keys.insert(index, value)
        leaf.children.insert(index, address)

        # fix up the node that is inserted
        self.__fixup(index_name, leaf, order)
```

第一步用到的函数与查找非常类似，因此不在此处详细介绍。第二步用到的函数也非常简单，不过多介绍。第三步用到的维护B+树的函数，其首先判断当前处理的结点是否需要分裂。如果是，判断其是不是根结点，如果是根结点，则需要新建一个根结点，并且将当前结点分裂、链接；如果不是根结点，则需要将当前结点分裂、链接后递归调用该函数。**通过递归来实现对整个B+树的维护**。伪代码如下：

```
    def __fixup(self, index_name, node, order):
        if node.is_full(order):
            # a new root needs to be created
            if node.parent == None:
                self.root = newroot = Node()
                node.parent = newroot
                newkey, lchild, rchild = self.__split(node, order)
                newroot.keys.append(newkey)
                newroot.children.append(lchild)
                newroot.children.append(rchild)
            else:
                # not a root
                parent = node.parent
                newkey, lchild, rchild = self.__split(node, order)
                index = parent.index_of_insert(newkey)
                parent.keys.insert(index, newkey)
                parent.children.insert(index+1, rchild)
                self.__fixup(index_name, parent, order)
        else:
            return
```

接下来看查找函数，通过索引来找到某个记录时，模块直接在文件中进行查找并返回目标记录的地址。这个过程类似于人脑查找B+树的过程，通过循环来不断向下查找，直接找到叶子结点。如果叶子结点中不包含目标值，则返回`False`，否则返回目标记录在文件中的地址。

```
    def search(self, name, value, type, length):
        keys = []
        children = []
        cur_bid = cur_offset = 0
        res_bid = res_offset = 0
        isleaf = False
        num = 0     # number of keys in this block
        while True:
            # build a node
            cur_block = self.buffer_manager.read_block(name, 1, cur_bid)
            # empty
            if empty:
                return False
            isleaf, num = struct.unpack('=?h', cur_block[:3])
```

```
            cur_offset = 3
            for i in range(num):
                rebuild the list for keys and the list for children
            # find the value
            if isleaf == False:
                go deeper according to comparison
            else:
                # find the value and return its position
                # the last child is the pointer
                if value not in keys[:num]:
                    return False
                else:
                    return the result
            keys = []
            children = []
            cur_offset = 0
```

最后看一看保存索引的方法。保存时首先清楚过去的索引文件，然后借助一个栈，依次将结点写入文件。写入`isleaf`与键值的数量后，交错写入孩子结点以及键值。全部保存完毕后，将该B+树释放以减轻内存的负担。伪代码如下：

```
    def save_Bplus(self, index_name, type, length):
        os.chdir(sys.path[0])
        # clear the previous index
        file = open('./index/'+index_name+'.ind', 'wb')
        file.close()
        stack = []
        stack.append(self.root)
        while len(stack) != 0:
            node = stack[0]
            pop the node
            bid = node.bid
            if node is not a leaf:
                push the children into the stack
                write isleaf and the number of keys
                for i in range(num):
                    write a child
                    write a key
                write the last child
                commit
            # a leaf
            else:
                write isleaf and the number of keys
                for i in range(num):
                    write a child
                    write a key
                write the pointer to the next leaf
                commit
        self.freeBplus()
```

测试部分，主要就是简单地调用编写的方法并查看结果。主要查看两点：**保存的文件内容是否正确**以及**能否正确地从文件中查询到结果**。测试代码如下：

```
if __name__ == '__main__':
    buffer_m = buffer.bufferManager()
    manager = index_manager(buffer_m)
    index_name = 'tt'
    values = ['aaa', 'cde', 'xhice', 'xsc', 'hidcu', 'xhsayi', 'xui', 'chausi']
    addresses = [[40, 6], [21, 8], [6, 6], [16, 23], [37, 21], [5, 24], [17, 3],
[34, 22]]
    manager.create_index(index_name, addresses, values, 4)
    type = '10s'
    length = 10
    manager.save_Bplus(index_name, type, length)
    # print(manager.search('test', 196, 'i', 4))
    print(manager.search('tt', 'xsc', '10s', 10))
    print(manager.search('tt', 'xxx', '10s', 10))
```

六、 遇到的问题及解决方法

1. Python的变量不定长，难以保存到二进制文件中
   为了方便，我们利用了`struct`包来将各变量的值转化为二进制，方便我们读写二进制文件。
2. 地址的表示问题
   文件中保存的指针应该是指向文件中的地址，而不能是内存地址。因此我们采取了block id + offset的表示方法，前者表示位于文件中的第几个块，后者表示内容在该块中的具体位置。当需要搜索某个记录，index manager获取搜索的键值，搜索完毕后将block id + offset的地址返回给API，API再调用record来读取该记录。当需要建立索引时，API首先要将所有记录及其地址读出，返回给API，API再调用建立索引的函数。
3. 代码统一的问题
   我们利用github来统一我们的开发进度。
4. 合作开发的问题
   整个开发过程，每个人的代码都有需要调整、修改的地方，以适应团队的开发。同时，我们设计的框架之下，也有许多细节之处需要我们不断地调整。我们的解决办法是保持沟通，用高效的沟通来统一共识，实现良好的团队开发。

七、 总结

本次实验的工作量不可谓不大，小组成员三人经历了大量的协商、沟通、编写代码，才一步步将工程开发到现在的规模。开发的过程加深了我们对课堂上学到的理论的理解，加强了我们团队合作的能力，让我们对数据库的开发有了更加透彻的理解，学会了处理许多细节。