MiniSQL 实验报告

3170105392 袁轶珂

1 实验目的

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 语句实现表的建立/删除；索引的建立/删除以及表记录的插入/删除/查找。

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。

2 实验分工

本实验由一人独立完成。

3 实验平台

开发语言：python 3.7

开发工具：pycharm

主要使用库：

prettytable	用于查询结果表的输出
re	正则表达式
prompt_toolkit	命令行工具包
struct	用于二进制文件存取

4 实验任务

4.1 实现功能

语法说明

支持标准的 SQL 语句格式，每一条 SQL 语句以分号结尾，一条 SQL 语句可写在一行或多行，要求所有的关键字都为小写。

创建表

该语句的格式如下:

```
create table 表名 (
    列名 类型 ,
    列名 类型 ,
    列名 类型 ,
    primary key ( 列名 )
);
```

支持三种基本数据类型，int，char(n)，float，n 在 1 到 255 之间。

设置数据类型为 unique 只需将 unique 写在该数据定义的最后, 如 [列名 类型 unique]。

主码 primary key 应写在括号内语句的最后，并且只能有不超过一个主码。

删除表

该语句的格式如下:

```
drop table 表名 ;
```

若该语句执行成功，则将表的信息、索引及记录全部删除。

创建索引

该语句的格式如下:

```
create index 索引名 on 表名 ( 列名 );
```

只能在已经创建过的表的基础上建立索引。

索引只能建立在表的单值属性上，即只有声明过 unique 的属性才能建立索引。

该语句执行成功后会在 index 中创建一棵 B+树，之后对表的数据添加和查询中会自动调用索引。

删除索引

该语句的语法如下:

```
drop index 索引名 ;
```

若该语句执行成功，则在表中删除该索引的信息，并将磁盘中的 B+树文件删除。

数据查询

该语句的语法如下:

```
select * from 表名 where 条件;
```

select 选择支持*，表示查询所有属性，若只查询部分属性，应键入相应属性名，中间用逗号隔开。

支持等值查询和区间查询，条件格式为 [列名 算术运算符 值] ，共支持 [= < > <= >=] 五种算术运算符。如果值的数据类型为 char, 则应以单引号括起表示其为字符串。

支持多条件查询，多个条件应在 where 语句中以 and 隔开。

若无查询条件，则 where 语句部分可以省略，程序输出表中所有值。

若该语句执行成功，则以表的形式输出符合语句查询条件的所有数据，第一行为表头属性名，其后按行输出数据，数据在表中的排列无顺序。

插入记录

该语句的语法如下：

```
insert into 表名 values ( 值1 , 值2 , … , 值n );
```

表名必须是已经存在的表。

插入值的顺序必须与表定义中属性的顺序完全一致。当值的数据类型为 char 时，应使用单引号括起以表示字符串。

若插入的值拥有 unique 属性则必须与其他值不同，否则会返回插入失败。

删除记录

该语句的语法如下：

```
delete from 表名 where 条件;
```

表名必须是已经存在的表。

Where 条件的使用方式与 select 语句相同，会将所有符合条件的记录删除。

如果没有删除条件，即要清空整张表的数据，可以省略 where 不写。

退出 MiniSQL 系统语句

该语句的语法如下：

```
quit;
```

输入后会出现退出成功的提示，buffer 中的相应记录也会进行保存以便下一次打开时读取。

5 具体实现

5.1 系统体系结构

MiniSQL 系统的具体体系结构如下：

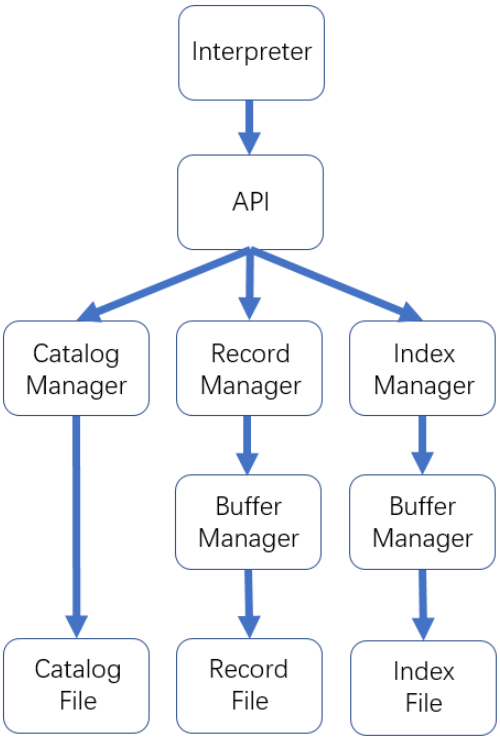


图 1 MiniSQL 设计结构

5.2 模块概述

5.2.1 Interpreter

Interpreter 模块直接与用户交互，主要实现以下功能：

1、 对用户输入的命令进行解释，

2、 生成命令的内部数据结构表示

3、 检查命令的语法正确性和语义正确性

4、 提供与 API 交互的接口

在本项目中，词法分析部分是利用正则表达式进行了实现。

读入时，一条语句相当于一个字符串，读入后通过提取少量的关键字判断该字符串归属于哪一类语句，调用相应的函数进行分析。

函数中即对该类语句的正则表达式格式进行了匹配，如果匹配成功则语法正确，解释器将语句中的信息分类写入相应的数据结构，词法分析完成；如果匹配失败，则返回报错信息，即匹配失败的部分不符合语法格式。

这种写法需要将所有语句的正则表达式格式进行实现，但好处是对于空格、换行的情况

的模糊匹配做的较好，满足更广泛的语句书写习惯。

此处以创建表的语句分析函数为例，在实例化该类之后，调用 new_table 函数即进入词法分析，若分析成功则该类成员即自 SQL 语句中分析出来的各变量结果。

```
class CreateT(object):
    def __init__(self):
        self.name = ''
        self.attr = {}
        self.pri = ''
        self.uni = []
        self.type = ['int', 'float']

    def new_table(self, sql):
        self.match_table(sql)
        self.match_attr(sql)

    def match_table(self, sql):
        match = re.search(r"table (\w+)\s?[(]", sql)
        if match:
            self.name = match.group(1)
        else:
            print("Syntax Error: Only table or index can be created")
            raise SQLException()

    def match_attr(self, sql):
        match = re.search(r"[(](.*)[)]", sql)
        if match:
            defList = match.group(1).split(',')
            for item in defList:
                matchAttr = re.search(r"(\w+)\s(\w+)\s?(\w+)?", item.strip())
                if matchAttr:
                    if matchAttr.group(1) == 'primary':
                        # handle primary key
                        matchPri = re.search(r"primary key [(](\w+)[)]", item.strip())
                        if matchPri and matchPri.group(1) in self.attr:
                            self.pri = matchPri.group(1)
                        else:
                            print("Error in using primary key : '%s'" % item.strip())
                            raise SQLException()
                    elif matchAttr.group(2) == 'char':
                        # handle attr in type char
                        matchChar = re.search(r"(\w+)\s(\w+)\s?[(](\d+)[)]\s?(\w+)?",
item.strip())
                        if matchChar:
```

```
                        self.attr[matchChar.group(1)] = matchChar.group(3)
                        if int(matchChar.group(3)) > 255 or
int(matchChar.group(3)) < 0:
                            print("Error: the size of variable in type char out of
domain")
                            raise SQLException()
                        if matchChar.group(4) == 'unique':
                            self.uni.append(matchChar.group(1))
                        elif matchChar.group(4) is not None:
                            print("Syntax Error near '%s'" % matchChar.group(4))
                            raise SQLException()
                    else:
                        print("Syntax Error near '%s'" % item.strip())
                        raise SQLException()
                elif matchAttr.group(2) in self.type:
                    # handle attr in type int or float
                    self.attr[matchAttr.group(1)] = matchAttr.group(2)
                    if matchAttr.group(3) == 'unique':
                        self.uni.append(matchAttr.group(1))
                    elif matchAttr.group(3) is not None:
                        print("Syntax Error near '%s'" % matchAttr.group(3))
                        raise SQLException()
                else:
                    print("Illegal varible type or misspelling of words")
                    raise SQLException()
            else:
                print("Syntax Error in '%s'" % item)
                raise SQLException()
        else:
            print("Syntax Error: lack of parenthesis")
            raise SQLException()
```

在本模块中，为更加方便的控制系统进程，在词法分析出现错误的时候抛出错误以结束当前进程。此处自定义类 SQLException 为词法分析错误，这样只需要在 commander 中 catch 该错误就可以使得命令循环一直执行下去。

```
    class SQLException(Exception):
    def __init__(self, err=''):
        Exception.__init__(self, err)
```

5.2.2 API

API 模块是整个系统的核心，其主要功能为提供执行 SQL 语句的接口。在该部分中对 Catalog Manager、Record Manager 和 Index Manager 提供的接口进行了统一调用。

在拿到 commander 传入的字符串后，首先进行初步的关键词解析，判断语句类别归属后调用 interpreter 模块的相应函数进行词法分析，分析得到的结果会调用各 Manager 的模块执行具体的操作。所以其主要函数 interpre 是一个类 switch case 结构，由于此处判断的情况比较复杂我利用 if else 进行了实现。（由于代码较长仅放部分代码为例）

```
def interpret(sql):
match = re.match("quit\s?;", sql)
if match:
    raise EOFError
sql = ' '.join(sql.split())
sig = re.match(r"(.*?) (.*?) ", sql)
if sig is None:
    print("Syntax Error")
    raise SQLException
op = sql.split(" ")[0]
ob = sql.split(" ")[1]
if op == 'create' and ob == 'table':
    c = CreateT()
    c.new_table(sql)
    t = createTable(c)
    t.save()
    newTable(t)
    print("Create Succeed")
elif op == 'create' and ob == 'index':
    indexName, tableName, attrName = createIndex(sql)
    createNewIndex(indexName, tableName, attrName)

elif op == 'drop' and ob == 'table':
    tableName = dropTable(sql)
    print(tableName)
    # judge if exist and remove from tables
    exist = False
    for i in range(len(tables)):
        if tableName == tables[i].name:
            exist = True
            del tables[i]
    if not exist:
        print("Drop Error : the table doesn't exist")
```

```
            raise SQLException()
        # remove from bufferList
        bufferList.dropBlock(tableName)
        # delete text
        os.remove('catalog\%s.txt' % tableName)
        os.remove('record\%s.txt' % tableName)
        print("Drop Success")
    ... ... ...
```

除以上主函数外在 API 模块中还有初始化函数。初始化函数的作用是对磁盘文件进行初始化，在接受到如创建表等指令时调用该函数，这样在 Manager 中只需要考虑处理而不需要考虑文件不存在的边界情况。并且，由于 API 模块拥有几乎所有接口，所以初始化时可以更加方便的将各模块的信息连接起来。此处以创建表函数代码为例：

```
def createTable(created):
    # duplicate of name judgement
    for item in tables:
        if created.name == item.name:
            print("Create Error : The table %s is already existed" % created.name)
            raise SQLException()
    # build table
    t = table(created.name, created.pri)
    for key in created.attr:
        if created.attr[key].isdigit():
            t.addColumn(key, 'char', len=int(created.attr[key]))
        else:
            t.addColumn(key, created.attr[key])
    for item in created.uni:
        t.setUnique(item)
    tables.append(t)
    return t
```

最后就是输出函数，在调用 Record Manager 的接口函数完成查询之后，应将结果以表的形式输出在命令行中，此处我们调用 prettyTable 库进行实现，该库的输出表格可选风格，默认风格即 mySQL 的表格输出风格。函数如下：

```
def printTableWidget(dictList):
    if len(dictList) == 0:
        print("Empty Set")
        return
    attrList = dictList[0].keys()
    tb = prettytable.PrettyTable()
```

```
    tb.field_names = attrList
    for item in dictList:
        tempList = list()
        for attr in attrList:
            tempList.append(item[attr])
        tb.add_row(tempList)
    print(tb)
```

5.2.3 Catalog Manager

Catalog Manager 负责管理数据库的所有模式信息，主要包括表和索引两部分。

表定义信息包括表的名称、属性列表、主码信息和该表的索引列表。属性列表中，每一项是一个字典，该字典包括四个键：属性名、数据类型、数据长度和是否为单值属性。索引列表的成员是在该模块中定义的索引类。

索引定义信息包含该索引的名称、该索引对应的属性信息（属性名、数据类型和数据长度）、该索引的归属表。

在全局中我们维护一个所有表组成的列表 tableList，每次进行执行语句命令时就在这个列表中对信息进行匹配。API 接口函数在新建表时会将实例化的新类 table 加入列表，所以匹配到表名之后就能得到所有相关属性的信息。

在该类中提供了很多接口方法供 API 模块调用，如数据一致性判断、生成对应属性正则表达式格式字符串等等，其实质上就是对表信息处理之后输出所需求的结果，在此不一一赘述。

每次本项目结束（quit）的时候，API 会遍历全局的 tableList，将所有表信息更新至磁盘。由于表定义信息不多，且属性个数有上限决定了表头并不会占用很大的空间，所以在存取的时候没有选择更加节省空间的二进制文件存取，而是选择了更加方便的 Json 格式存取。具体做法是将所有的类成员作为值写入字典，键则是该类成员的名称。这样只需要借助 json 库就能方便的实现表定义的读取，索引信息的读取也同理。

```
    def save(self):
    fileDict = dict()
    fileDict['name'] = self.name
    fileDict['primaryKey'] = self.__primaryKey
    fileDict['column'] = self.__col
    fileDict['size'] = self.__size
    fileDict['blockNum'] = self.blockNum
    tempList = list()
```

```
    for item in self.indexList:
        indexInfoList = [item.indexName, item.attrName, item.type, item.len]
        tempList.append(indexInfoList)
    fileDict['index'] = tempList
    file = open('catalog\%s.txt' % self.name, 'w')
    file.write(json.dumps(fileDict))
    file.close()

def read(self, fileName):
    fileDict = dict()
    file = open('catalog\%s' % fileName, 'r')
    data = json.loads(file.read())
    file.close()
    self.name = data['name']
    self.__primaryKey = data['primaryKey']
    self.__col = data['column']
    self.__size = data['size']
    self.blockNum = data['blockNum']
    tempList = data['index']
    for item in tempList:
        ind = index(item[0], item[1], self.name, item[2], item[3])
        self.indexList.append(ind)
    # self.printTable()
```

5.2.4 Record Manager

Record Manager 负责管理表的数据，通过 Buffer Manager 对 record file 进行访问、增加、删除等操作。

数据在磁盘中的存取基本单位是块(block)，本系统中设置 block 的大小为 4Kb。在存储记录时就要考虑到块的情况对数据进行一定的包装，所以设计记录文件的组织形式如下图所示：

<table>
<tr><th></th><th>数据类型</th><th>长度（byte）</th><th>含义</th></tr>
<tr><td>文件头</td><td>int</td><td>4</td><td>总块数</td></tr>
<tr><td rowspan="12">块信息</td><td>int</td><td>4</td><td>freeList长度L</td></tr>
<tr><td>int</td><td>L * 4</td><td>freeList</td></tr>
<tr><td>int..</td><td rowspan="3">size(record)</td><td rowspan="3">记录1</td></tr>
<tr><td>char..</td></tr>
<tr><td>float..</td></tr>
<tr><td>int</td><td>4</td><td>保留位（通常为0）</td></tr>
<tr><td>int..</td><td rowspan="3">size(record)</td><td rowspan="3">记录2</td></tr>
<tr><td>char..</td></tr>
<tr><td>float..</td></tr>
<tr><td>.</td><td>.</td><td>.</td></tr>
<tr><td>.</td><td>.</td><td>.</td></tr>
<tr><td>.</td><td>.</td><td>.</td></tr>
</table>

图 2 记录文件组织格式

文件头为一个 int，表示该表下的记录文件中共有多少个块，以便添加新块的时候利用指针跳转快速寻址。

在文件头之后即每个块的数据，一个块分为 freeList 和 record 两部分。FreeList 保存了该块中空余未写数据的空间的序号。比如该块最多可写三条数据，已经在第二条数据的位置写入了数据，则 freeList 长度为 2，内容为 0 2。在每次新写入数据的时候根据 freeList 寻址即可快速找到写入数据的位置，以这种方式存储块中的数据是无序的。在 freeList 之后就是存放的各条 record，每条 record 之后存了一个 int，值为 0，作为保留位。每条 record 的大小为 4 字节的整数倍以方便磁盘读取。

Record Manager 内提供了相应的函数来实现对 record file 的相应操作。由于数据以块的形式存储在硬盘中，所以以记录为单位的操作应在块的基础上进行，调用 Buffer Manager 提供的块操作接口方法实现。在每次接受到 SQL 语句所含变量之后，对数据一致性进行检测，如表名是否存在和是否符合单值属性特点等，如果符合，则对符合条件的 buffer 进行操作，如果当前块不能满足要求则调用 Buffer Manager 函数替换新块。此处以插入记录代码为例：

```
def newRecord(tableName, record):
# check tableName
find = False
for i in range(len(CatalogManager.tables)):
    if CatalogManager.tables[i].name == tableName:
        t = CatalogManager.tables[i]
```

```
            find = True
            break
    if not find:
        print("Insert Error : the table doesn't exist")
        raise SQLException()
    # check the integrity of record
    result = t.checkIntegrity(record)
    # check unique
    uniqueList = t.getUniqueList()
    if len(uniqueList) > 0:
        cond = list()
        for item in uniqueList:
            tempList = [item['name'], '=', result[item['index']]]
            cond.append(tempList)
        print("Condition generated in unique checking:", cond)
        selectRes = traversalRecord(t, cond)
        if len(selectRes) > 0:
            print("Insert Error : the value is not unique")
            raise SQLException()
    # build No list for query
    Nolist = list(range(1, t.blockNum + 1))
    # write record
    find = False
    l = bufferList.list
    for i in range(len(l)):
        if tableName == l[i].table.name:
            Nolist.remove(l[i].blockNo)
            checked = l[i].addRecord(result)
            if checked:
                print("Insert Succeed in buffer list")
                blockNo = l[i].blockNo
                find = True
                break
    # if there isn't appropriate block, update block
    if not find:
        for i in range(len(Nolist)):
            bufferList.insertNewBlock(t, Nolist[i])
            checked = bufferList.list[0].addRecord(result)
            if checked:
                print("Insert Succeed in new buffer existed")
                blockNo = Nolist[i]
                find = True
                break
    # if all the block is full, add new
```

```
    if not find:
        newRecordBlock(t)
        t.blockNum += 1
        bufferList.insertNewBlock(t, t.blockNum)
        checked = bufferList.list[0].addRecord(result)
        if checked:
            print("Insert Succeed in new building buffer")
            blockNo = bufferList.list[0].blockNo
    return result, blockNo
```

在本系统中，对数据的操作共有插入、删除、访问三种，插入和删除实质上为 buffer 上的读写其思路是十分相似的，而访问略有不同，在访问时还需要借助 Index 来提升访问效率。Index 的具体组织方式将在报告的 Index Manager 部分中进行详细阐释，在此处只简单分析 index 在查询中所起作用的部分。

在 API 接口将分析后的语句内容传入时，Record Manager 将对查询条件进行分析，如果查询条件中有建立过索引的属性，则利用索引进行查询，反之则采用遍历的方式。

遍历的方式十分简单，将该表对应文件的所有块依次读入 buffer 中，遍历所有数据，对每一条数据判断是否符合条件，完成访问。

当有索引的时候，我们首先根据条件对 B+树进行访问。以区间查询为例，当查询大于某值时，则在 B+树中访问该值，找到对应的叶节点后，依据指针访问向右的所有叶节点内的键值对，根据 B+树中指针的信息访问文件 block，可以省去很多 block 的访问。普通遍历和带索引遍历的代码如下：

```
    def traversalRecordWithIndex(t, cond, blockList):
    res = list()
    # find in bufferList
    l = bufferList.list
    for i in range(len(l)):
        if l[i].table.name == t.name and l[i].blockNo in blockList:
            blockList.remove(l[i].blockNo)
            for j in range(l[i].recordMaxNum):
                if j not in l[i].freeList:
                    tempDict = checkOneRecord(t, l[i].recordList[j], cond)
                    if tempDict is not None:
                        res.append(tempDict)
            bufferList.updateBlock(i)
    # load the other blocks
    for i in blockList:
        bufferList.insertNewBlock(t, i)
```

```
        for j in range(l[0].recordMaxNum):
            if j not in l[0].freeList:
                tempDict = checkOneRecord(t, l[0].recordList[j], cond)
                if tempDict is not None:
                    res.append(tempDict)
    print("I'm the filtered result:", res)
    return res


def traversalRecord(t, cond, delete=False, withBlockNo=False):
    print("I'm in traversal, success will coming soon")
    res = list()
    Nolist = list(range(1, t.blockNum + 1))
    # find in bufferList
    l = bufferList.list
    for i in range(len(l)):
        if l[i].table.name == t.name:
            Nolist.remove(l[i].blockNo)
            for j in range(l[i].recordMaxNum):
                if j not in l[i].freeList:
                    tempDict = checkOneRecord(t, l[i].recordList[j], cond)
                    if tempDict is not None:
                        if delete:
                            l[i].deleteRecord(j)
                        else:
                            if withBlockNo:
                                tempDict['BlockNo'] = l[i].blockNo
                            res.append(tempDict)
            bufferList.updateBlock(i)
    # load the other blocks
    for i in range(len(Nolist)):
        bufferList.insertNewBlock(t, Nolist[i])
        for j in range(l[0].recordMaxNum):
            if j not in l[0].freeList:
                tempDict = checkOneRecord(t, l[0].recordList[j], cond)
                if tempDict is not None:
                    if delete:
                        l[0].deleteRecord(j)
                    else:
                        if withBlockNo:
                            tempDict['BlockNo'] = l[0].blockNo
                        res.append(tempDict)
    if delete:
```

```
        return True
    return res
```

5.2.5 Index Manager

Index Manager 负责 B+树索引的实现，并根据指令对 B+树进行增删等相应的维护。

在本实验中，由于 Index 的块定义与 Record 相差较大，所以将 Index 的 buffer 交互部分写入了 B+树的数据结构，每次结点的访问是一个磁盘 IO 的过程，在分析具体数据结构之前，首先描述 index file 的定义：

	数据类型	长度（byte）	含义
文件头	int	4	根节点地址
block0	int	4	freeList[0]
	int	4	freeList[1]
	…		
block1	int	4	leaf & size
	type	n	key
	int	4	value(指针)
	type	n	key
…	…	…	…

图 3 index file 结构

首先是文件头，文件头为一个 int 表示根节点的地址，由于在文件中结点是乱序的所以要依靠文件头找到根节点实现树的访问。Block0 是固定的信息头，是与 record 类似的 freeList 列表。表示文件中哪些块是空的可以被写入，freeList 是一个有序数组，最后一个值为当前块的最大值。自 block1 向后即索引记录部分，每个块都是 B+树的一个结点。块中信息以一个 int 开头，这个 int 是指示位，包含是否为叶节点和结点内数据量两个信息。如果是叶节点，则该位信息为 2 * size + 1 为奇数，如果是非叶节点则该位为 2 * size 为偶数。在指示位之后即结点内的键值对，共有 size 个。每个 block 内 value 的数量比 key 多一个，指示向同层的下一个结点，方便对树的遍历。每个索引都有对应的文件，每次访问一个结点，从根节点开始，根据结点内的指针找到下一个结点的地址并进行访问，重复以上步骤即可访问 B+树的所有结点。

B+树的节点类 BPlusNode，由于 B+树要求树结点大小与块大小一致，所以 BPlusNode 即维护一个与块大小相同的树节点的信息，其类成员有该 B+树叉数、索引名、该块在文件中的指针以及该结点是否为根节点。结点信息的读取即按照文件组织结构将该点的类成员信息写入对应块的位置即可，以 save 代码为例：

```
def save(self):
    if not self.__dirty:
        return
    # write back
    fileName = 'index\%s_%s.txt' % (self.__index.indexName, self.__index.tableName)
    file = open(fileName, 'rb+')
    file.seek(self.myPos, os.SEEK_SET)
    print("I'm in %d, saving " % file.tell(), self.recordList)
    recordLen = int((len(self.recordList) + 1) / 2)
    if self.__isLeaf:
        b = struct.pack("i", recordLen * 2 + 1)
    else:
        b = struct.pack("i", recordLen * 2)
    file.write(b)
    # write record
    writeList = self.recordList
    writeList.append(self.__index.emptyRecord()[1])
    pat = self.__index.pattern()
    for i in range(0, len(writeList), 2):
        b = struct.pack(pat, writeList[i], writeList[i+1])
        file.write(b)
    file.close()
```

B+树类 BPlusTree，其类成员有索引定义信息（来自 Catalog Manager 接口），树的叉数、和当前节点（BPlusNode 类）。在每次调用 B+树时，只需要实例化 BPlusTree 类即可，其构造函数会读入当前 B+树的根节点作为 Node。

B+树的访问较为简单，BPlusNode 类提供了 checkChild 方法，返回待访问值所在的子节点指针，B+树对当前结点进行保存，然后读入下一个结点，直到访问到叶节点为止，具体代码如下：

```
def gotoLeaf(self, key):
    fatherList = list()
    while True:
        toPos = self.node.checkChild(key)
        if toPos == 0:
            break
        fatherList.append(self.node.myPos)
        self.node.save()
        self.node = BPlusNode(self.index, self.forkNum, toPos)
    return fatherList
```

B+树的键值插入比较复杂，由于在创建索引时已经建立了一个空结点作为根节点，左移不需要考虑空树的情况。只需要根据是否为叶节点两种情况进行分析即可。

如果是叶节点，且当前结点内键值对的个数小于 M-1，则直接进行插入即可。如果结点内键值对个数超过 M-1，则将当前结点分裂为左右两个节点，将右侧节点的 key 写入父节点中，则当前结点转移到父节点中，进入非叶结点的讨论。

对于非叶结点的情况，若键值对的个数小于 M-1 则插入结束。若结点内个数超过 M-1，则将当前结点分裂为左右两个结点，与叶节点不同的是需要进位一个 key 和两个指针，分别指向左右结点。

在 BPlusNode 类中提供了将当前结点分裂为两个结点的方法，返回新创建的结点，代码如下：

```
def splitNode(self):
    self.__dirty = True
    resList = list()
    if self.__isLeaf:
        for i in range(2 * self.__min, len(self.recordList)):
            resList.insert(0, self.recordList.pop())
        self.recordList.append(0)
    else:
        for i in range(2 * self.__min + 1, len(self.recordList)):
            resList.insert(0, self.recordList.pop())
        resList.insert(0, self.recordList[-1])
    return resList
```

B+树的删除是本次实验的难点，如果删除后键值对的个数大于等于(M-1)/2 上取整，则删除结束。如果键值对的个数小于下界，则先检查兄弟结点的情况，如果兄弟结点中的值有富余，则向兄弟节点借一个值添加至自己结点，并视左右结点的情况将父结点中的 key 进行替换。如果兄弟结点都没有多余的键值对，则将该结点与左右兄弟任一结点进行合并，生成新结点，并删除父节点中的 key。

在删除父节点中的 key 之后，重新检查父节点中的键值对个数，如果小于下界，则重复以上借结点和合并两步。不同的是在借结点后，需要将父节点的 key 写入当前节点，而将兄弟节点的 key 写入父节点。如果是合并的情况，则不仅要合并兄弟节点还要将下移的父节点 key 一并合并到当前结点中。此处以叶节点的部分代码为例：

```
def borrowFromBrother(self, brotherPos, relation):
    if brotherPos == 0:
```

```
        return False, 0, 0
    brotherNode = BPlusNode(self.index, self.forkNum, brotherPos)
    if not brotherNode.checkExtraKey():
        return False, 0, 0
    # have extra Key
    brotherNode.setDirty()
    self.node.setDirty()
    if relation == 'left':
        oldKey = self.node.recordList[1]
        brotherNode.recordList.pop()
        self.node.recordList.insert(0, brotherNode.recordList.pop())
        self.node.recordList.insert(0, brotherNode.recordList.pop())
        brotherNode.recordList.append(self.node.myPos)
        borrowKey = self.node.recordList[1]
    else:
        oldKey = brotherNode.recordList[1]
        self.node.recordList.pop()
        self.node.recordList.append(brotherNode.recordList.pop(0))
        self.node.recordList.append(brotherNode.recordList.pop(0))
        self.node.recordList.append(brotherNode.myPos)
        borrowKey = brotherNode.recordList[1]
    brotherNode.save()
    self.node.save()
    return True, oldKey, borrowKey
def dropIndex(self, key):
    fatherList = self.gotoLeaf(key)
    # check if exist
    checked = self.node.checkExistKey(key)
    if not checked:
        self.node.save()
        self.node = BPlusNode(self.index, self.forkNum, 0, isRoot=True)
        return False
    # check if extra
    checked = self.node.checkExtraKey()
    self.node.deleteRecord(key)
    if checked:
        self.node.save()
        self.node = BPlusNode(self.index, self.forkNum, 0, isRoot=True)
        return True
    # find its brothers
    if len(fatherList) == 0:
        # is root
        self.node.save()
        return True
```

```
    # is not root
    myPos = self.node.myPos
    fatherPos = fatherList[-1]
    fatherNode = BPlusNode(self.index, self.forkNum, fatherPos)
    leftPos, rightPos = 0, 0
    for i in range(0, len(fatherNode.recordList), 2):
        if fatherNode.recordList[i] == myPos:
            if i - 2 > 0:
                leftPos = fatherNode.recordList[i - 2]
            if i + 2 < len(fatherNode.recordList):
                rightPos = fatherNode.recordList[i + 2]
            break
    # try to borrow left brother's record
    checked, oldKey, newKey = self.borrowFromBrother(leftPos, 'left')
    if checked:
        # use position to locate key and change it
        for i in range(1, len(fatherNode.recordList), 2):
            if fatherNode.recordList[i] == oldKey:
                fatherNode.recordList[i] = newKey
                break
        fatherNode.setDirty()
        fatherNode.save()
        self.node = BPlusNode(self.index, self.forkNum, 0, isRoot=True)
        return True
```

5.2.6 Buffer Manager

Buffer Manager 负责缓冲区的管理，主要功能有：

1、 根据需要，读取指定的数据到系统缓冲区或将缓冲区中的数据写出到文件

2、 实现缓冲区的 LRU 替换算法，当缓冲区满时选择合适的页进行替换

3、 记录缓冲区中各页的状态，如是否被修改过等

4、 为提高磁盘 I/O 操作的效率，缓冲区与文件系统交互的单位是块，块的大小应为文件系统与磁盘交互单位的整数倍。

在本实验中实现的 buffer 是较为简单的版本，不支持记录的跨块储存，并且每条记录作了 4 字节的对齐，所以会有一定程度的空间浪费，但读写和管理会较为方便。记录文件的存储模式已在报告的 Record Manager 部分给出了，buffer 会按照相应的格式将一个块的信息从磁盘中读入内存，读入代码如下：

```
def read(self, No):
    self.blockNo = No
```

```
    fileName = "record\%s.txt" % self.table.name
    file = open(fileName, 'rb')
    file.seek(4, os.SEEK_CUR)
    for i in range(0, No - 1):
        file.seek(4, os.SEEK_CUR)
        b = file.read(4)
        offset, = struct.unpack('i', b)
        file.seek(offset, os.SEEK_CUR)
    # just for test
    b = file.read(4)
    cur, = struct.unpack('i', b)
    print("I'm read block No : ", cur)
    b = file.seek(4, os.SEEK_CUR)
    # read free list
    b = file.read(4)
    freeLen, = struct.unpack('i', b)
    pattern = ''
    for i in range(freeLen):
        pattern += 'i'
    b = file.read(4 * freeLen)
    temp = struct.unpack(pattern, b)
    self.freeList = list(temp)
    # print(self.freeList)
    if self.recordMaxNum - freeLen != 0:
        print('im moving pointer in free list')
        file.seek((self.recordMaxNum - freeLen) * 4, os.SEEK_CUR)
    # print(file.tell())
    # read Record
    pattern = self.table.pattern()
    for i in range(self.recordMaxNum):
        b = file.read(self.table.getSize())
        temp = struct.unpack(pattern, b)
        self.recordList[i] = list(temp[:-1])
    self.recordList = self.table.charDecoding(self.recordList)
    # print(self.recordList)
    file.close()
```

每次 record 被更新时，都会在 buffer 中进行相应的改变并将 buffer 的 dirty 指示符设为 true，而磁盘中暂时不作改变。只有当该块被替换或系统关闭时才会将 buffer 重新存入内存，即采用延时存储的模式管理 buffer。

而 buffer 的替换算法采用了 LRU，即维护一个列表类，列表的成员是当前在内存中的所有 buffer。每次调用 buffer 进行改动的时候即该 buffer 活跃，调用 update 方法将该 buffer

放到列表的最前端。每次在列表需要替换块时调用 pop 函数释放最后一个 buffer 再载入新块，每次块被释放的时候会调用该块的 save 函数对磁盘进行更新以保持数据的一致性。

```
class LRUList(object):
    def __init__(self):
        self.list = []

    def updateBlock(self, i):
        # move to the head
        self.list.insert(0, self.list.pop(i))

    def insertNewBlock(self, table, No):
        if len(self.list) == BLOCK_MAX_NUM:
            self.popBlock()
        # a new block
        r = recordBlock(table)
        r.read(No)
        self.list.insert(0, r)

    def dropBlock(self, tableName):
        for i in range(len(self.list)):
            if self.list[i].table.name == tableName:
                self.list.pop(i)

    def popBlock(self):
        l = self.list.pop()
        l.save()
```

5.2.7 Commander

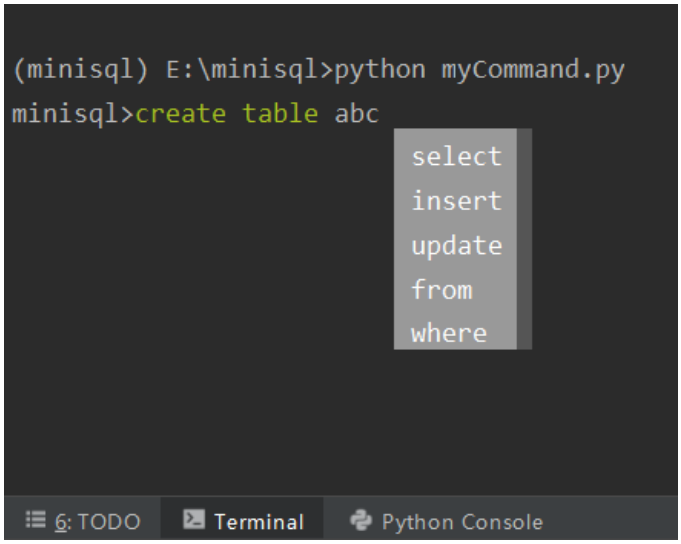
Commander 模块即程序的起始模块，该模块维护了一个命令行循环，此处借助了 prompt 库实现了一个有提示符、高亮语法和历史命令储存的命令循环。每次以判断分号作为一条语句的结尾，并调用 API 的主函数执行该语句，并对词法分析时抛出的异常进行捕捉。主函数代码如下：

```
def main():
    API.readTables()
    print(API.tables)
    session = PromptSession(lexer=PygmentsLexer(SqlLexer))
    end = True
    res = ''
    while True:
        try:
```

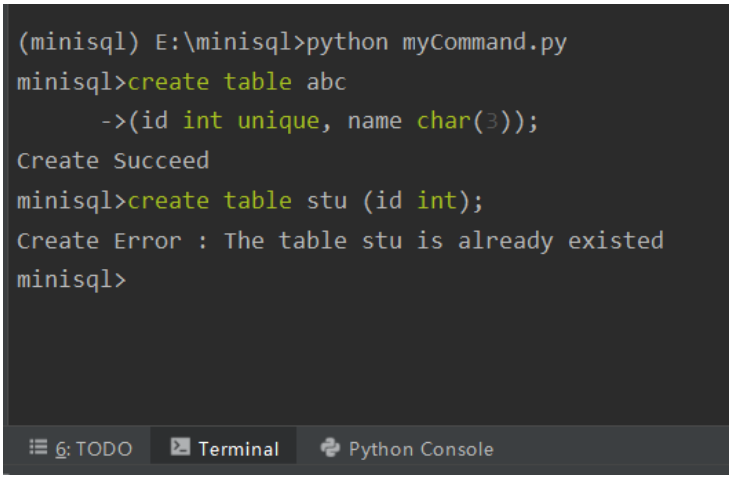
```
            if end:
                text = session.prompt('minisql>', completer=sql_completer)
            else:
                text = session.prompt('      ->', completer=sql_completer)
        except KeyboardInterrupt:
            end = True
            res = ''
            continue
        except EOFError:
            break
        else:
            text = text.strip()
            res += text
            if res[-1] == ';':
                end = True
                # handle res
                try:
                    API.interpret(res)
                    res = ''
                except API.SQLException:
                    res = ''
                    continue
                except EOFError:
                    break
            else:
                end = False
    API.saveTables()
    API.bufferList.saveAllBuffer()
    print('GoodBye!')
```

6 实验结果

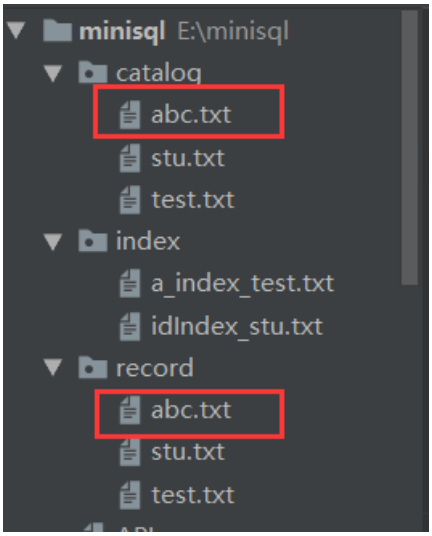
命令行的初始界面如下，在键入语句时有提示和高亮效果：



创建一张新表，如果创建成功会返回成功提示，如果失败会返回错误信息：



可以从文件组织中看到，在表创建后相应文件已经建立：



对刚建立的表进行数据插入，当插入的数据出现错误的时候会出现提示信息：

```
minisql>insert into abc values(1, 'bbb');
Insert Success
minisql>insert into abc values(2, 'ccc');
Insert Success
minisql>insert into abc values(3, 'ddd');
Insert Success
minisql>insert into abc values(4, 'ee');
Insert Success
minisql>insert into abc values(5, 'f');
Insert Success
minisql>insert into abc values(5, 'fff');
Insert Error : the value is not unique
```

对插入的数据进行查询检测（区间查询和等值查询）：

```
minisql>select * from abc where id < 5;
+----+------+
| id | name |
+----+------+
| 4  |  ee  |
| 3  | ddd  |
| 2  | ccc  |
| 0  | aaa  |
| 1  | bbb  |
+----+------+
minisql>select * from abc where name = 'aaa';
+----+------+
| id | name |
+----+------+
| 0  | aaa  |
+----+------+
minisql>
```

对多条件的连接查询的结果：

```
minisql>select id from abc where id > 4 and name = 'aaa';
Empty Set
minisql>select id from abc where id > 4 and name = 'f';
+----+
| id |
+----+
| 5  |
+----+
minisql>
```

将记录从表中删除：

```
minisql>delete from abc where id = 0;
Delete Succeed
Delete Success
minisql>select * from abc;
+----+------+
| id | name |
+----+------+
| 1  | bbb  |
| 2  | ccc  |
| 3  | ddd  |
| 4  |  ee  |
| 5  |  f   |
+----+------+
minisql>
```

为了对索引进行测试，我们首先将abc的数据量扩充到100，然后为了突出索引的效果，我们将 buffer 的大小从 4K 改为 16byte，每个块中只有一条数据，并对 LRU 块数量进行调整。最后加入时间的输出。

在未加入索引前：

```
minisql>select * from abc where id >= 95;
+----+------+
| id | name |
+----+------+
| 96 |  ca  |
| 97 |  cb  |
| 98 | cc1  |
| 99 | cdd  |
+----+------+
Query OK, time 0.14s
```

在创建索引后：（name 不为 unique 属性不能建索引）

```
minisql>create index idIndex on abc (name);
Create Error : attribute is not unique
minisql>create index idIndex on abc (id);
```

查询结果:

```
minisql>select * from abc where id >= 95;
+----+------+
| id | name |
+----+------+
| 98 | cc1  |
| 96 |  ca  |
| 99 | cdd  |
| 97 |  cb  |
+----+------+
Query OK, time 0.03s
minisql>
```

由于查询方式不同，所以对块的访问顺序也不同，用遍历的方式和用索引的方式查询到的结果顺序也不同。在本测试中由于改变了块的大小，没有体现除数据的局部相邻性在索引查询中的优势，当数据量更大时，如果数据都在同一个块或几个块内，在访问块数上索引查询的优势将会更加放大，速度优势也会更加明显。