数据库系统结构 实验报告

BY 刘宇淏 2019011306

1 运行说明

本项目采用Python编写。 需要至少Python3.9的环境来进行运行以满足项目中采用的类型标注语法。 除此之外， 还需要安装相应的依赖项， 这包括了PTable， prompt_toolkit， tqdm，antlr4-python3-runtime，numpy等。安装完毕后可以启动项目，建议通过requirements.txt来进行安装。

其中main.py是入口文件。包含一定的命令行选项：

- -d，–debug，这个参数表示是否开启调试信息。参数为True和False，默认为False
- -f，–file [filename]，表示静默运行中运行的文件。如果采用这个参数，将进行非交互式运行；通常这个选项用于导入数据。

2 系统结构设计

本项目采用经典的数据库系统结构设计， 分为以下几个模块： 文件系统（fio.py）， 记录系统（recordsystem）， 元信息系统（metasystem）， 索引系统（indexsystem）， 查询系统（querysystem）和数据库管理系统（dbsystem），其中还有一个sqlsystem其实是parser。

其中数据库系统负责以上前四个模块的统合管理，并通过相应对SQL语句的解析调用不同模块的不同方法。而查询系统则负责的是针对单表的操作，包括索引的管理，select单表查询，drop，insert等相关操作。索引系统系统了针对B-Tree索引的实现，记录系统则管理了文件上所有的记录。文件系统实现了页式文件系统，使得可以缓存一定的页来加快文件的I/O。

2.1 文件系统

文件系统采用了页式文件系统，以8kb作为一个页一次性完整的进行读写。由于Python的二进制交互能力并不是特别充分，所以采用了numpy的ndarray作为一页的载体。

对于页的交换算法，则采用LRU算法，当某一页被访问的时候将该页推到链表的最前面，当某一页由于缺页被读入的时候将该页放在链表的最前面。当链表比整个Cache的大小要大的时候则将链表的最后一项重新写回到文件中。

主要接口如下：

- get_page(self, fid: int, pid: int) -> np.ndarray 获得某一页
- put_page(self, fid: int, pid: int, page: np.ndarray) 写回某一页
- open_file(self, filename: str) -> int 打开某个文件
- close_file(self, fid: int) 关闭打开的某个文件

2.2 记录系统

记录系统包括了记录文件和记录文件的简单管理（不进行详细概述）。记录文件以页式文件作为基础，采用了定长的记录管理方式。针对三种不同类型的数据，其格式分别如下：

- INT：8字节，其中低4字节是数据，高4字节代表是否为NULL
- FLOAT：8字节，其中低4字节是数据，高4字节该表是否为NULL
- VARCHAR(N)：N字节，其中不区分空字符串和NULL

而在记录文件的第一页，则保存了整个记录文件的一些元信息：

- record_length：单条记录的长度（根据上面的加和）
- records_per_page：每页的记录条数
- page_count：总共有多少页
- record_count：总共有多少记录
- next_page：下一个空闲页的标号
- bitmap_length：每页页头位图的长度

针对每一页的设计，说明如下：

- next_page：4字节，表示空闲页组成的空闲链表。如果这个值和自身的页号相同表示自身不是空闲页
- bitmap：records_per_page / 4字节，位图，记录了该页每个槽位的占用情况
- records：记录

整体的算法比较明朗了，通过伪代码进行呈现：

```
def insert(self, record: Record):
    page = allocate_empty_page()
    rid = page.insert(record)
    if page.full:
        self.next_page = page.next_page
        page.next_page = page.page_id
    return rid

def drop(self, rid: RID):
    page = get_page(rid.page_id)
    page.mark_free(rid.slot_id)
    if page.next_page = page.page_id:
        page.next_page = self.next_page
        self.next_page = page.page_id
```

这里简单的说明了插入和删除的算法，大概能表明实现的意思。

接下来简述相应的接口说明（RecordHandle类）：

- insert(self, record: Record) -> RID 插入一条记录
- update(self, rid : RID, record: Record) 更新记录
- pop(self, rid: RID) 删除记录

2.3 索引系统

索引系统采用了B-Tree进行实现。每个B-Tree的Page对应于一个节点。其中分为内部节点和叶子节点。只有叶子节点记录了RID是多少。内部节点的每个槽位记录的是（Key，LeafNode）的元组，而叶子节点则记录了（Key，RID）的元组。

其中类BTreeNode是两种节点的基类。而InternalNode和LeafNode分别实现了两种节点。为了方便起见，本次实现并不像一般的B-Tree一样有哨兵，而是Key和Value是一一对应的。这使得在插入和删除的时候需要一些小技巧：也就是孩子需要给自己的父亲返回自己在修改后的最小Key来让父亲更新指向自己的指针的Key值。这会在伪代码中进行说明。

其中索引只实现了针对INT类型的索引。并把NULL置为了一个充足小的值来保证和上层一般行为一致（NULL比谁都小）。接下来将进行伪代码的说明：

```
def insert(self, key: int, value: RID):
    self.root.insert(key, value)
    if self.root.need_split():
        self.split_root()

# internalnode
def insert(self, key: int, value: RID):
    insert_leaf = self.key_lower_bound(key)
    update_key = self.vals[insert_leaf].insert(key, value)
    self.keys[insert_leaf] = update_key
    if self.vals[insert_leaf].need_split():
        self.split(insert_leaf)

# leafnode
def insert(self, key: int, value: RID):
    id = self.lower_bound(key)
    self.keys.insert(id, key)
    self.vals.insert(id, value)
    return self.keys[0]
```

从上面的伪代码可以大概理解什么是更新自己最小的键值。 删除操作也是类似于传统的B-Tree但是同样有这样更新的操作。如果发现自己的节点被删光了，则返回None来表示自己是空页，父亲节点可以从自己的孩子中将其删除。

主要接口如下：

- insert(self, key: int, value: RID) 插入一条
- remove(self, key: int, value: RID) 删除一条
- select(self, operator: str, key: int) -> set[RID] 通过operator提供的条件来进行选择（operator可以是>; <; >=; <=; =; <>

同时管理索引还提供了建立索引的功能。

2.4 元信息系统

该系统记录了Table，Database等元信息，通过pickle模块进行序列化和反序列化。这个meta文件也是一个数据库目录下标识这是一个数据库的标志。

通过元信息，实现了Convert类来帮助从二进制序列化、反序列化一个可读可比较的记录。本部分并没有太多接口需要说明，在反序列化之后很多信息直接从类中读取即可，类似于数据记录。

2.5 查询系统

该系统负责将单表查询的条件汇总，并进行单表的Select操作。通过将不同的condition获得的结果取交集，就能够得到一定条件下对表的Select结果。同时，也提供了Select RID的方法在不需要转换的时候加速查询。

其中条件是通过一个list[tuple[str, str, Any]]的方式传入，通过lambda的方式进行组合。条件的具体格式是(col, op, value)，首先通过Operator类的compare方法来获得op对应的比较函数，然后通过闭包捕获到列表：list[tuple[int, (Any, Any) -> bool, Any]]中，格式是(col_id, compare, value），然后进行比较的时候：

```
def compare(x):
    for col_id, cmp, value in conditions:
        if cmp(x[col_id], value):
```

```
                return True
        return False
```

通过build_condition函数完成了上述的功能；这个功能也是这个系统中最为重要的功能。其他的drop和update都是基于这个功能实现的。

除了Select功能以为，查询系统还系统了对Value的类型检查来保证Value的类型是符合元信息中所规定的类型的，也同样进行了Primary Key的检查工作。

相关重要接口：

- select(self, conditions: list[tuple[str, str, Any]]) -> list[list[Any]] 选择记录
- select_rid(self, conditions: list[tuple[str, str, Any]]) -> set[RID] 选择RID
- insert(self, values: list[Any]) 插入单条
- value_typecheck(self, values: list[Any]) -> list[Any] 单条类型检查
- drop(self, conditions: list[tuple[str, str, Any]]) 删除记录
- update(self, setter: list[tuple[str, Any]], conditions: list[tuple[str, str, Any]]) 单表更新

2.6 数据库管理系统

最终，以上的系统通过数据库管理系统统合。该系统提供了DBVisitor这个类，在遍历语法树的同时进行以上系统的调用。

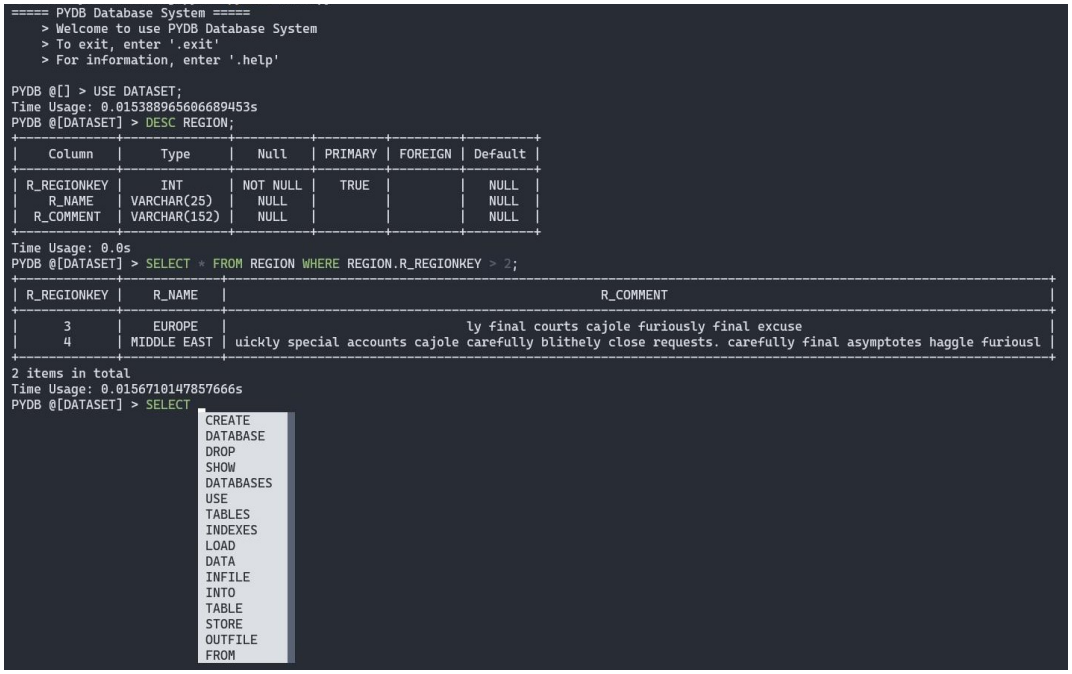
实现中采用了一个文件夹对应于一个数据库的方式，其中的.meta文件是标识数据库的标志。

同时这个系统还进行了一些其他的检查，比如Foreign Key的约束，多表Join等。其中Join采用了谓词下推，将不是Join的条件先进行Select然后再进行Join来完成加速。

这个部分只有一个接口，就是run(self, sql: str) -> QueryResult表示运行SQL查询语句。

3 实现结果

本次实现提供了带自动提示，美观的报错信息的CLI界面：



本次实验完成了所要求的基本功能外，还实现了多表join功能。

分工：solo