程序框架说明 

> By 李易非 3160105705
>
> 在本次实验中，我负责程序框架设计，实现 API, CatalogManager, BufferManager, RecordManager

一、设计简介

本次工程中需要实现 Interpreter，API，Record Manager，Catalog Manager，Buffer Manager，Index Manager；

- Interpreter
- API 是程序的核心，它直接接收 Interpreter 提供的数据信息，调用各个 Manager 来完成数据库的插入 / 删除 / 更新；
- Record Manager 管理各个表的具体记录；
- Catalog Manager 管理数据库的元信息；
- Buffer Manager 管理数据库与文件层的交互；
- Index Manager 管理数据库中的索引 (B+ 树) ；

  各个 Manager 相互合作，共同维护数据库的数据；

二、重要类简介

在工程中，各个 Manager 的具体介绍可详见各 Manager 部分，这个部分主要介绍一些其它的重要类；

3.1 MiniSQL

工程中面临的一大难题是 Manager 之间的互相调用，以及 API 对于各个 Manager 的调用。我们的解决方案是实现一个 MiniSQL 类，它拥有各个 Manager 的静态成员变量，这样做的理由如下：

- 各个 Manager 并非 API 的成员变量，也并非 Interpreter 的成员变量，要在不同函数内部调用不同 Manager 是一件棘手的事情；
- 整个程序共享一份 Manager，一定程度上反映了 **全局性**

  具体类定义如下：

```
class MiniSQL
{
    private:
    static API & api;
    static CatalogManager & catalog_manager;
    static RecordManager & record_manager;
```

```
    static BufferManager & buffer_manager;
    static IndexManager & index_manager;


    public:
    static API & get_api();
    static CatalogManager & get_catalog_manager();
    static RecordManager & get_record_manager();
    static BufferManager & get_buffer_manager();
    static IndexManager & get_index_manager();

};
```

- 使用范例
  - `API & api = MiniSQL::get_api()`，Interpreter 可以正常调用 API ,实现对应功能；
  - `CatalogManager & catalogmanager= MiniSQL::get_catalog_manager()`, API或者其它需要元数据的 Manager 可以轻松调用 Catalog Manager，提取需要信息；

3.2 Method

工程中面临的一大难题是对于数据文件的读写，数据库文件都是二进制文件，不同的解释方式决定了不同的信息，我们实现了 `Method` 类，其中包含整个数据库都需要的函数；

具体类定义如下：(简化版)

```
class Method
{
    public:
    //这个函数内部有new

    static int rawdata2int(const char * rawdata);
    static float rawdata2float(const char * rawdata);
    static void rawdata2string(const char * rawdata, int length, string &
out_str);
    static void string2rawdata(const string & str, const int type, char *
rawdata);
    static bool isSatisfyConditon(const char * rawdata, const int cond,
const string & operand, const int type);
    static const int getLengthFromType(int type);
};
```

具体说明

- `rawdata2int` 将一段 rawdata 翻译为整型变量；

- `rawdata2float` 将一段 rawdata 翻译为 float 型变量；
- `rawdata2string` 将一段 rawdata 翻译为 string;
- `string2rawdata` 将 string 翻译为 rawdata;
- `isSatisfyCondition` 给定 rawdata 与它的类型，判断这段record是否符合对应的条件与操作数；
- `getLengthFromType(type)`
  - 介绍这个函数之前，有必要介绍工程中对数据类型的处理，利用宏定义
    - `#define TYPE_INT 256`
    - `#define TYPE_FLOAT 257`
    - `#define TYPE_CHAR 255`
    - `#define MIN_TYPE_LENGTH 4`
  - 因为 varchar 类型变量最长为 255 bytes，小于255的类型一定是 varchar 类型变量，此时它的 data_type 就是它的实际长度；
  - 因为 free_list 的维护至少需要 4 bytes来标记下一条删除记录的位置，所以最短的类型长度为4，varchar(j) j < 4 均被看作 varchar(4)
  - Int / float 正常翻译即可；

```
switch(type)
    case TYPE_INT : 长度 = 4;
    case TYPE_FLOAT : 长度 = 4;
    default :
        if(type < 4)
            长度 = 4;
        else
            长度 = type;
```

3.3 Attribute

Attribute 的信息对于数据库至关重要，Attribute 类中含有 属性名称，属性类型，属性长度，是否 Primary, 是否 Unique；

```
class Attribute
{
    string name;
    short type;
    int length;
    bool isPrimary;
    bool isUnique;

};
```

3.4 Table

Table 类存储在 CatalogManger 中，通过 CatalogManger 的 get_table 函数，数据库可以获得一个表的所有信息；

```
class Table
{
    string table_name;
    int record_length;                                  // 记录的逻辑长度
    unordered_map <string, Attribute> Name2Attri; //名字到属性的hash
    unordered_map <string, int > Name2Pos;  //属性名字到属性位置的hash
};
```

3.5 Block

Block 类中包含文件名，Block ID，Block内容信息；它由 BufferManager 进行集中管理；

```
class Block
{
    string filename;            // Block 所在的文件
    int block_id;               // Block 在文件中的位置
    bool dirty;                 // Block 是否被修改过
    char * content;             // Block 中的真实内容

};
```

三、文件存储方式

3.1 设计简介

本次实验中，有3种类型的文件

1. 表的 metadata；
2. 索引的 metadata;
3. 数据文件；
4. 索引文件；

   前三种文件我们进行了**同质化处理**，可以用相同的处理方式来读对应数据，第四种文件因为 B+ 树的特性，采取了不同的存储方式；

3.2 具体实现

3.2.1 数据文件基本约定

3.2.1.1 物理存储说明

- 采用**定长记录**方法，`varchar(n)` 将在数据文件中分配 n 个字节的空间，不会因为具体记录长度的变化而变化；
- 采用 **Heap File** 方法，即文件内部 record 没有固定的存储顺序，通过 **free list** 来标记所有被删除的记录。free list header 在文件中占据4位 (int)，若为 -1 则说明到了 free list 尾部，若不为 1，则代表下一条无效 record 在文件中的**绝对地址**。其余被删除记录的前四位与上述逻辑相同，若为 -1 则说明到了 free list 尾部，若不为1，则代表下一条无效 record 在文件中的**绝对地址**；
- 每一条数据后新增一个 **有效位** ，在遍历文件过程中，仅仅通过 **free list** 来判断一个 record 是否被删除非常耗时，最终我们决定 **空间换时间**， 在每一条记录后都添加一个有效位，若为1则表示有效记录，若为0则表示无效记录；
- 记录 **不会跨块存储** ，在一个 Block 不足以容纳一条记录时，直接前往下一个 Block；

3.2.1.2 逻辑存储说明

- 每个数据文件都有自己的 metadata，存储在所有文件的 **第一个 Block** 中，metadata包含
  - Record Length (物理长度，包含了有效位)
  - free list header，free list 链表头，指向一个被删除的记录；
  - Record Count，代表文件中记录总数，注意这里包含无效记录。这是为了确认文件 **EOF** 的具体位置；
- 其余的 Block 正常存储数据，若为无效记录，则前四位指向 free list 下一条无效记录；若为有效记录，则是一个 record 的正常存储单元；

3.2.1.3 注意事项

- 逻辑 Record Length 为物理 Record Length 减1；
- 因为要维护一个 free list，我们为每一个指针都分配了四个字节的空间，所以对于 varchar(j) (j < 4) 类型，我们都按照 varchar (4) 进行处理；
- Record Count，代表文件中记录总数，注意这里包含无效记录。这是为了确认文件 **EOF** 的具体

位置；

3.2.2 具体数据文件存储内容

3.2.2.1 Table Meta 存储说明

Table Meta 存储在特定文件夹 `TableMeta/` 中，其中包含一个提供所有 Table 名字的 `TableMeta/tables` 文件，以及各个 Table 对应的 Meta Data 文件 `TableMeta/Table`

- `TableMeta/tables` 存储数据库中所有表名 (32 bytes) ；
- `TableMeta/Table` 存储特定 table 的属性信息，文件中的一条记录存储属性名 (32 bytes) 、数据类型 (2 bytes) 、是否主键 (1 byte) 、是否唯一 (1 byte) ；

  把 Table Meta 又分成 tables 与 特定 table 是为了保持同一个文件的 **record length 一致**，因为不同表的属性个数是不同的，如果存储在一起，很难在不浪费空间的前提下保持记录长度的一致性；

3.2.2.2 Index Meta 存储说明

Index Meta 存储在特定文件夹 `IndexMeta/` 中，其中包含一个提供所有 Index 信息的 `IndexMeta/indices` 文件。

- `IndexMeta/indices` 存储 索引名 (64 bytes) 、表名 (32 bytes)、属性名 (32 bytes)；

👇是一个例子

```
+--- TableMeta
|        |--- tables
|        |--- book
|        |--- student
|
+--- IndexMeta
         |--- indices
```

3.2.2.3 具体记录存储说明

具体记录文件存储在特定文件夹 `data/` 下，每一张表占据一个文件，存储表中真实的记录。

👇是一个例子

```
+--- data
        |--- book
        |--- student
```