



浙江大学

MiniSQL设计报告

第12组

课程名称： 数据库系统

指导教师： 庄越挺

2020 年 6 月 30 日

- 一、实验概览
 - 1.1 实验目的
 - 1.2 实验需求
 - 数据类型
 - 表定义
 - 索引的建立和删除
 - 查找记录
 - 插入和删除记录
 - 1.3 实验环境
 - 1.4 整体框架
- 二、各模块实现功能
 - 2.1 Interpreter
 - 2.2 API
 - 2.3 Catalog Manager
 - 2.4 Record Manager
 - 2.5 Index Manager
 - 2.6 Buffer Manager
- 三、分工说明
- 四、各模块提供接口与内部实现
 - 4.1 Interpreter
 - 4.1.1 `Interpreter` 类
 - 4.1.2 工作流程
 - 4.1.3 输入说明
 - 4.1.4 输出说明
 - 4.2 API
 - `API` 类
 - 4.3 Catalog Manager
 - `CatalogManager` 类
 - 4.4 Record Manager
 - `RecordManager` 类
 - 4.5 Index Manager
 - 4.5.1 `IndexManager` 类
 - 4.5.2 B+节点类
 - 4.5.3 B+树类
 - 4.6 Buffer Manager
 - 4.6.1 `Page` 类
 - 4.6.2 `BufferManager` 类
- 五、界面说明
- 六、系统测试

一、实验概览

1.1 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。

1.2 实验需求

数据类型

要求支持3种数据类型: `int`, `float`, `char(n)`; 其中 $1 \leq n \leq 255$

表定义

一个表最多可以定义32个属性, 各属性可以指定是否为 `unique`; 支持单属性的主键定义

索引的建立和删除

对于表的主属性自动建立B+树索引, 对于声明为 `unique` 的属性可以通过SQL语句由用户指定建立/删除B+树索引 (所有的B+树索引都是单属性单值的)

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

插入和删除记录

支持每次一条或多条记录的插入操作; 支持每次一条或多条记录的删除操作

1.3 实验环境

本项目全部采用C++语言实现, 通过[GitHub](#)托管代码、管理版本。

1.4 整体框架

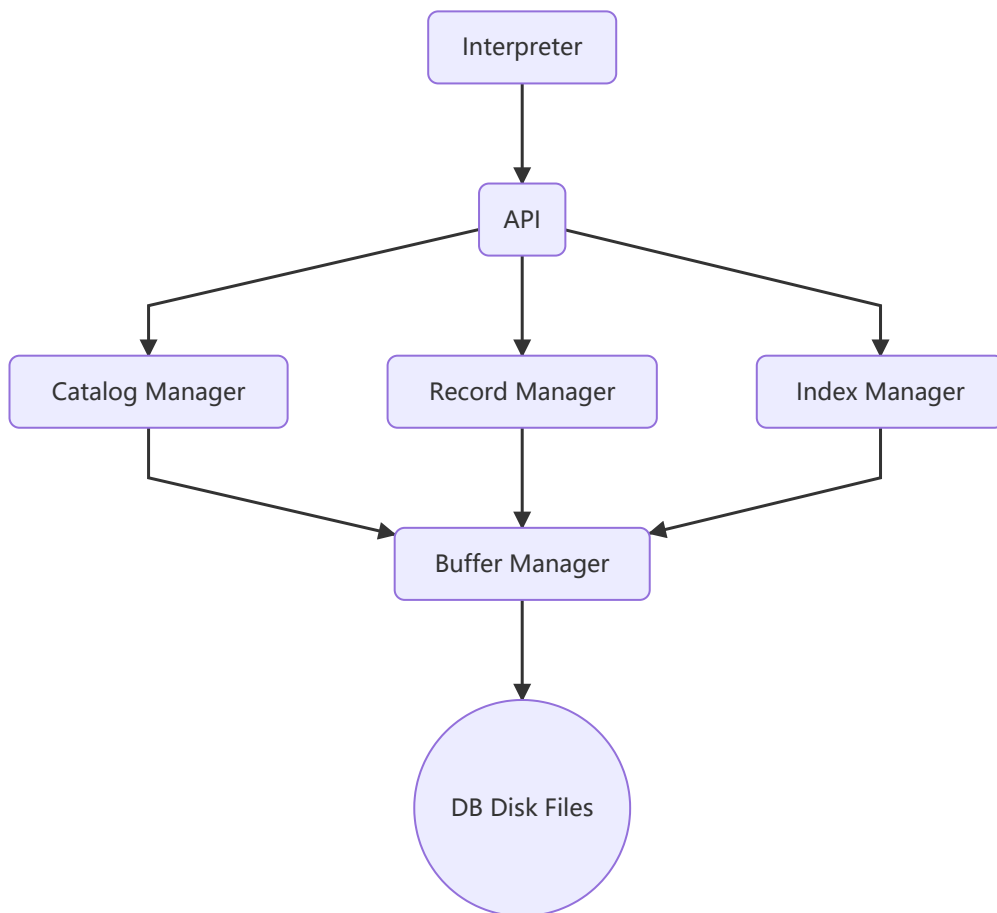
MiniSQL主要包含6个模块:

- Interpreter
- API
- Catalog Manager
- Record Manager
- Index Manager
- Buffer Manager

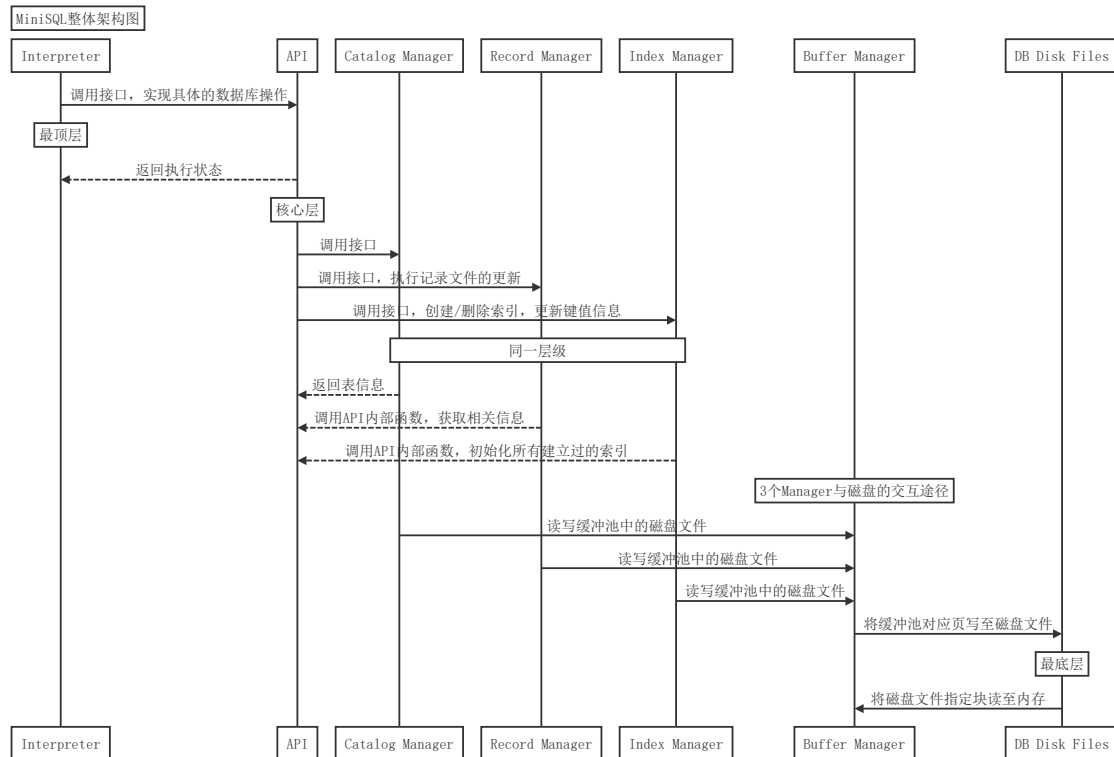
此外, 数据库在磁盘上保存至相应的磁盘文件中:

- `./database/catalog` 文件保存数据库目录信息
- `./database/record/"` 目录下保存各张表的记录文件
- `./database/index/"` 目录下保存各张表的各个索引文件

MiniSQL的各个模块的层级图如下:



各模块间的详细交互方式如下图所示：



详见第二章"各模块实现功能"。

二、各模块实现功能

2.1 Interpreter

Interpreter模块负责解析SQL语句，并返回相应信息。主要实现了程序流程控制整体框架，即“启动并初始化、接收命令、处理命令、显示命令结果、循环、退出”流程。同时接收并解释用户输入的命令，生成命令的内部数据结构表示，检查命令的语法正确性和语义正确性，按照要求返回相应信息

2.2 API

API模块整合了更底层的Record Manager、Index Manager、Catalog Manager，以实现Interpreter层需要的更为复杂的功能。作为数据库系统的核心，API模块提供给Interpreter的接口包括：创建表、删除表、创建表、删除表、查询记录、插入记录、删除记录。API模块综合调度3个Manager；从Catalog Manager获取表/数据库的信息，进一步调用Record Manager和Index Manager，完成记录和索引的更新或查询。

2.3 Catalog Manager

Catalog Manager主要记录了所有表的定义信息，将每一个表信息按一定格式记录在文件中，并提供方位上述信息的接口，同时Catalog Manager调用Buffer Manager实际完成表的创建、删除等等相关功能。

2.4 Record Manager

Record Manager主要为API提供记录的插入、删除、查询功能的接口；Record Manager负责每个表的记录文件的创建、更新以及在磁盘上的保存。利用Buffer Manager，Record Manager可以有效地通过较少的磁盘访问进行尽可能多的记录读写。

2.5 Index Manager

Index Manager实际上是B+树的驱动模块，负责索引创建、索引删除、键值插入、键值删除、键值等值查找。

B+树是平衡树数据结构，通常用于数据库和操作系统的文件系统中，其插入与修改具有较稳定的对数时间复杂度，因为数据访问时磁盘I/O耗时更加显著，B+树可以最大化内部结点的储存数据量，从而显著减少I/O耗时。在文件"BpTree.h"中，用类模板的方式实现了B+树的查找、插入、删除操作；同时，利用Buffer Manager，读取磁盘上的索引文件可以重构B+树，在磁盘上进行记录至索引文件可以保存B+树。

Index Manager掌管3张由索引文件名映射到B+树地址的 `std::map`，分别对应键值为 `int`、`float`、`string` (即 `char(n)`) 类型。IndexManager 对象可以间接地创建、删除或找到索引对应的B+树，再进行之前提到的索引相关的操作。Index Manager实现了索引文件/B+树在磁盘上的记录与读取，即“永久”索引。

2.6 Buffer Manager

具体来说，Buffer Manager模块涉及 `Page` 和 `BufferManager` 对象，并实现了相应部分功能。功能如下：

1. 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. 记录缓冲区中各页的状态，如是否被修改过、是否有效等。
3. 提供缓冲区页的pin功能，及锁定缓冲区的页，不允许替换出去。(用于替换策略)
4. 使用时钟替换算法实现页面替换。

Buffer Manager掌管内存中的一片缓冲池。更高层的Catalog Manager、Record Manager、Index Manager利用Buffer Manager与磁盘文件交互，是一种能有效减少磁盘访问次数的读写数据库文件的方法。

三、分工说明

- 谢廷浩：负责部分Buffer Manager模块、部分API模块、Index Manager驱动部分的设计、B+树与磁盘交互的实现，并进行了全局的debug，将各个模块串联起来。
- 黄仁泓：定义Page类，完成部分BufferManager函数；定义Interpreter类，完成创建删除表格、索引、执行文件等功能，实现MiniSQL界面等

四、各模块提供接口与内部实现

4.1 Interpreter

4.1.1 Interpreter 类

```
1  class Interpreter
2  {
3  public:
4      API *api;
5      string filename;
6      int interpreter(string s);
7      Interpreter() {};
8      ~Interpreter() {};
9  private:
10     string getword(string s, int& index);
11     string getword_no_symbol(string s, int& index);
12     int string2num(string s);
13 };
```

成员变量

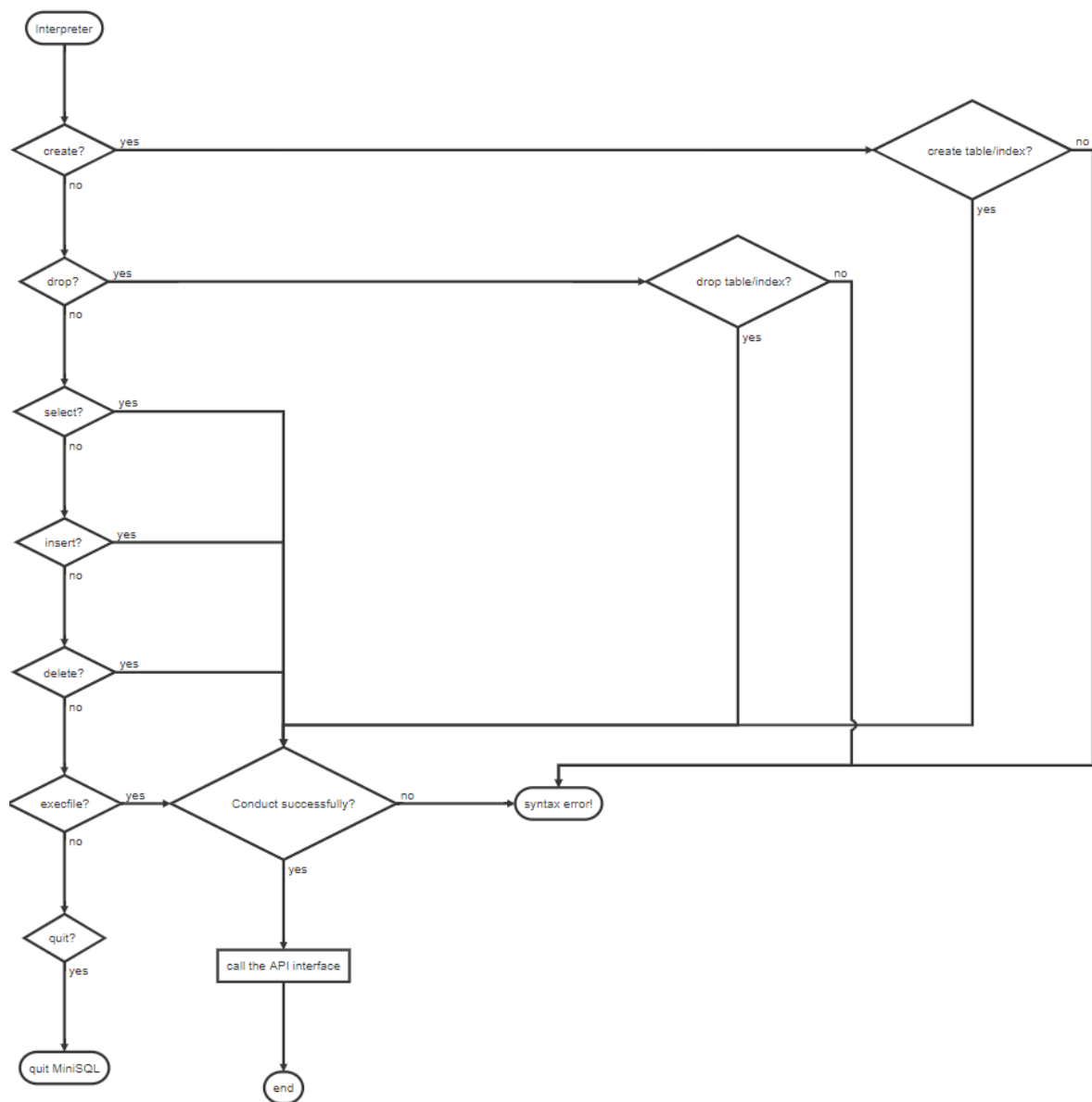
- API接口 `api`
- 执行文件名 `filename`

成员函数

- 构造函数和析构函数
- 解析函数(用于语义解析传入的语句) `interpreter`
- 提取单词或字符 `getword`
- 只读取单词忽略标点符号 `getword_no_symbol`

4.1.2 工作流程

Interpreter的工作流程图如下：



4.1.3 输入说明

- 输入的所有字符都不区分大小写，在输出时会自动转为小写状态
- 换行前的语句后面要加至少一个空格/回车。比如：
`select sno·换行·from student;` 是不合理的，而 `select sno ·换行·from student;` 是合理的
- 输入语句若需要声明是字符串数据，需要用单引号扩起来，比如 `'example_string'`
- 支持标准的sql语句

(1) 建表

```

1 create table [表名] {
2     [属性名1] [数据类型1] [可选择说明是否为unique],
3     [属性名2] [数据类型2] [可选择说明是否为unique],
4     ... ..
5     primary key (属性名x) [必须存在]
6 };
  
```

其中数据类型有三种：整数类型 `int`、浮点数类型 `float`、字符串类型 `char(n)`

(2) 创建索引

```
1 create index [索引名]
2 on [表名] ([表中某一属性名]);
```

(3) 删除索引

```
1 drop index [索引名] on [表名];
```

(4) 删除表格

```
1 drop table [表名];
```

(5) select-from语句：对表格所有的元组做投影

```
1 select [属性名1], [属性名2], [属性名3], ...
2 from [表名];
```

(6) select-from-where语句：选出部分表格的元组做投影

```
1 select [属性名1], [属性名2], [属性名3], ... ...
2 from [表名]
3 where [条件1] and [条件2] and ... ...
```

其中条件为关于属性的不等式，比如： `sage >= 30`, `sgender <> 'M'`, `sname = 'A'`，并且只支持条件之间的逻辑与操作（即关键字 `and`）

如果希望全选，则用 `select *`

(7) delete-from语句：全表记录/元组删除

```
1 delete from [表名];
```

(8) delete-from-where语句：删除满足条件的记录/元组

```
1 delete from [表名]
2 where [条件1] and [条件2] and ... ...
```

- 不同的语句之间以分号 ; 作为分隔，在输入sql语句时，也只在碰到分号 ; 后才开始执行
- 可以在运行程序后在命令行直接输入sql语句，或是全部写在文件中 `sql.txt`，然后在命令行输入 `execfile sql.txt` 后执行

4.1.4 输出说明

- 如果SQL语句执行成功，执行后会返回成功信息。

(1) 建表/删除表格

```
1 Create table [表名] successfully!
2 Drop table [表名] successfully!
```

(2) 建立索引/删除索引


```
1 Create index [索引名] on table [表名] successfully!
2 Drop index [索引名] on table [表名] successfully!
```

(3)select语句

若查询结果不为空，则输出第一行为属性名，其余每行表示一条记录。

```
1 select result: #####
2      [选择的属性名]
3      [符合条件的记录]
4 #####
```

若查询结果为空

```
1 select result: #####
2 [选择的属性名]
3 #####
```

(4)insert语句

```
1 insert result:#####
2 insert [插入记录的数目] record(s) into [表名] successfully!
3 #####
```

(5)delete语句

```
1 delete result:#####
2 delete [输出记录的数目] record(s) on [表名] successfully!
3 #####
```

(6)quit语句

```
1 Exit program!
```

- 如果SQL语句执行失败，执行后返回失败原因。

(1)建表失败可能出现的原因

```
1 empty table!
2 create table syntax error!
3 char size overflow!
4 unknown data type
5 primary key not exist!
6 Table [表名] already exists!
```

(2)建索引失败可能出现的原因

```
1 index not exist error!
2 create index syntax error!
3 Table [表名] not exists!
4 Index [索引名] already exists!
5 Indices on table [表名] full!
6 Attribute [属性名] not exists
```

(3) 删除表或索引失败可能出现的原因

```
1 empty table!
2 index not exist error!
3 drop index syntax error!
4 Table [表名] not exists!
5 Index [索引名] on table [表名] not exists!
```

(4) select语句失败可能出现的原因

```
1 query on multiple tables is not allowed!
2 empty table!
3 invalid operator!
4 table [表名] not exist!
5 attributes being queried not exist!
```

(5) insert语句失败可能出现的原因

```
1 insert syntax error!
2 table [表名] not exist!
3 attributes being queried not exist!
```

(6) delete语句失败可能出现的原因

```
1 delete syntax error!
2 table [表名] not exist!
```

4.2 API

API模块封装在 API 类中。

API 类

```
1 class API
2 {
3     private:
4         /*Internal function members below*/
5         /*一些内部函数，不应该被interpreter调用！
6         可能会被API本身、catalog manager、record manager、index manager模块调用
7         没有列全，用到时自行补足即可*/
8         //To be continued.
9
10        //tableName:表名称
11        //recordContent:记录record中实际内容
12        //recordResult:用于存储一条record内容的指针
13        //将recordContent中的内容转为一个char*指向的内容，之后recordResult指针会作为其它
14        函数参数
15        void recordStringGet(string tableName, vector<string>* recordContent,
16        char* recordResult);
17    public:
18        /*下面3个指针在main模块中才初始化，避免和3个manager产生冲突*/
19        //指向全局catalog manager的指针
20        catalogManager *cm;
21        //指向全局record manager的指针
```

```

20     RecordManager *rm;
21     //指向全局index manager的指针
22     IndexManager *im;
23
24     /*外部接口*/
25     API() {};
26     ~API() {};
27     //创建表
28     bool createTable(std::string table_name, Attribute attribute, int
primary, Index index);
29     //删除表
30     bool dropTable(std::string table_name);
31     //创建索引
32     void createIndex(string index_name, string table_name, string
attribute_name);
33     //删除索引
34     void dropIndex(string table_name, string index_name);
35     //展示记录 -- 无条件
36     int showRecord(string table_name, vector<string>* attribute_names);
37     //展示记录 -- 有条件
38     int showRecord(string table_name, vector<string>* attribute_names,
vector<Condition>* conditions);
39     //插入记录
40     void insertRecord(string table_name, vector<string>* record_content);
41     //删除记录 -- 无条件
42     int deleteRecord(string table_name);
43     //删除记录 -- 有条件
44     int deleteRecord(string table_name, vector<Condition>* conditions);
45
46     //获取recordSize的函数
47     int recordSizeGet(string tableName);
48     //获取表中SingleAttribute的函数
49     int attributeGet(string tableName, vector<SingleAttribute>
*attributeVector);
50
51
52     /*Internal function members below*/
53     /*一些内部函数，不应该被interpreter调用！
54     可能会被API本身、catalog manager、record manager、index manager模块调用
55     没有列全，用到时自行补足即可*/
56     //内部函数，插入**单个数据**对应的索引
57     void insertIndex(string table_name, string index_name, char* key_addr,
int type, int blockID);
58     //内部函数，删除**单个数据**对应的索引
59     void deleteIndex(string table_name, string index_name, char* key_addr,
int type);
60     //内部函数，删除给定地址开始的**一条记录**对应的索引
61     void deleteRecordIndex(string table_name, char* record_begin, int
record_size, Attribute attributes);
62     //内部函数，插入给定地址开始的**一条记录**对应的索引
63     void insertRecordIndex(string table_name, char* record_begin, int
record_size, Attribute attributes, int blockID);
64     //获得所有index的信息: <index文件名, index的type> -- 在Index_Manager.cpp中用
于初始化所有index文件
65     vector<pair<string, int> > allIndexInfoGet();
66     //To be continued.
67 };

```

成员变量

- `cm`: Catalog Manager的指针
- `rm`: Record Manager的指针
- `im`: Index Manager的指针

成员函数

- `bool createTable(std::string table_name, Attribute attribute, int primary, Index index)`

创建表接口；创建名为 `table_name` 的表，其属性保存在 `attribute` 结构中，主键指定为第 `primary` 个，并传入一个空的 `index` 结构。内部依次调用Catalog Manager和Record Manager的建表接口，实现表的创建。

- `bool dropTable(std::string table_name)`

删除表接口；删除名为 `table_name` 的表。内部先删除表的所有索引，再依次调用Catalog Manager和Record Manager的删除表接口，实现表的删除。

- `void createIndex(string index_name, string table_name, string attribute_name)`

创建索引接口；在 `table_name` 表的 `attribute_name` 属性上创建名为 `index_name` 的索引（**注意：允许不同表拥有同名索引**）。内部先调用Catalog Manager的 `CreateIndex()` 在catalog上记录新建的索引，再利用Record Manager的 `createIndex()` 新建索引文件，之后调用Index Manager的 `createIndex()` 创建相应的B+树，最后通过 `RecordManager::indexRecordAllAlreadyInsert()` 将已经存在的所有记录对应的（键值，blockID）插入到B+树中。

- `void dropIndex(string table_name, string index_name)`

删除索引接口；删除 `table_name` 表上的 `index_name` 索引。内部先调用Catalog Manager的 `DropIndex()` 在catalog上删除表的索引信息，随后依次调用Record Manager、Index Manager的 `dropIndex()` 接口，删除磁盘以及内存中的索引文件/B+树。

- `int showRecord(string table_name, vector<string>* attribute_names)`

无条件查询记录接口（即没有 `where` 语句）；将 `table_name` 表的**所有记录**投影到 `attribute_names` 上进行打印。实际上会交由有条件查询记录接口进行查询。

- `int showRecord(string table_name, vector<string>* attribute_names, vector<Condition>* conditions)`

有条件查询记录接口（即有 `where` 语句）；将 `table_name` 表中所有符合 `conditions` 中所有条件的记录，投影到 `attribute_names` 上进行打印（**注意：只支持 and 连接的条件**）。如果 `conditions` 的某个条件为**等值判定**，且判定属性已经建立了索引（那就意味着如果存在符合条件的记录，该记录必然是唯一的），则会利用索引直接找到该记录所在的块，再呈递给Record Manager加速查询；否则Record Manager需要遍历磁盘文件的每个块找到所有符合条件的记录。

- `void insertRecord(string table_name, vector<string>* record_content)`

插入记录接口；在 `table_name` 表中插入一条记录，记录内容由 `record_content` 指向的 `std::vector` 记录（依次保存每个属性的字符串值）。内部先处理 `record_content`，将 `int`、`float` 类型的数据的字符串转换成二进制数据，暂时存放在内存的 `recordString`（记录插入的一条记录）中。随后调用Record Manager将 `recordString` 写到缓冲池中，最终记录到磁盘上。此外，如果该表建立了索引，则需要向索引文件插入相应的（键值，blockID）以更新索引。

- `int deleteRecord(string table_name)`

无条件删除记录接口（即没有 `where` 语句）；删除 `table_name` 表的**所有记录**。实际上会交由有条件删除记录接口进行删除。

- `int deleteRecord(string table_name, vector<Condition>* conditions)`

有条件删除记录接口（即有 where 语句）；删除 table_name 表中同时所有满足 conditions 中每个条件的记录（注意：**只支持 and 连接的条件**）。如果 conditions 的某个条件为**等值判定**，且判定属性已经建立了索引（那就意味着如果存在符合条件的记录，该记录必然是唯一的），则会利用索引直接找到该记录所在的块，再呈递给Record Manager加速删除；否则Record Manager需要遍历磁盘文件的每个块找到所有符合条件的记录并删除。

4.3 Catalog Manager

Catalog Manager模块将表的创建、删除功能实现的相关代码封装在 CatalogManager 类中，相对应的代码"CatalogManager.h"和"CatalogManager.cpp"中，此外还提供了提供了一些方便API调用的接口。

CatalogManager 类

```

1  class CatalogManager{
2  public:
3      BufferManager &BM;
4      CatalogManager(BufferManager& b):BM(b),tablelist()
5      {
6          init();
7      }
8      void init();
9      ~CatalogManager(){}
10     /*
11     @parameter: table_name,attribute_name,primary_key,index
12     Function: insert a table in catalog file
13     Output: if repeat, output error table_not_exist
14     */
15     void CreateTable(std::string t_name, Attribute attribute, int primary,
Index index);
16     /*
17     @parameter: table_name
18     Function: delete a table in catalog file
19     Output: if no such table, output error table_not_exist
20     */
21     void DropTable(std::string t_name);
22     /*
23     @parameter: table_name,attribute_name,index
24     Function: Create a index for an attribute on a table
25     Output: if no such table, output error table_not_exist.
26     if no such attribute, output error:attribute_not_exist.
27     if index repeat, output error index_exist
28     */
29     void CreateIndex(std::string t_name, std::string a_name, std::string
i_name);
30     /*
31     @parameter: table_name,index
32     Function: find attribute through index
33     Output: if no such table, output error table_not_exist
34             if no such index, output error index_not_exist
35     */
36     std::string IndextoAttr(std::string t_name, std::string i_name);
37     /*
38     @parameter: table_name,index
39     Function: Delete index of a table

```

```

40     output: if no such table, output error table_not_exist
41             if no such attribute, output error attribute_not_exist
42             if no such index, output error index_not_exist
43     */
44     void DropIndex(std::string t_name, std::string i_name);
45     /*
46     @parameter: table_name
47     Function: check if the table exists, ouput true for yes and false for no
48     */
49     bool IsTable(std::string t_name);
50     /*
51     @parameter: table_name, attribute_name
52     Function: check if the attribute exists, true for yes and false for no
53     Output: if no such attribute,output error table_not_exist
54     */
55     bool IsAttribute(std::string t_name , std::string a_name);
56     /*
57     @parameter: table_name
58     Function: Get the attribute from a named table
59     Ouput: if no such table, output error table_not_exist
60     */
61     Attribute GetAttribute(std::string t_name);
62     /*
63     @parameter: table_name
64     Function: show table information
65     output: if no such table, output error able_not_exist
66     */
67     void PrintTable(std::string t_name);
68     /*get the index of the table*/
69     Index GetIndex(std::string t_name);
70     /*Get all table name*/
71     std::vector <std::string> GetAllTable();
72 private:
73     /*tranform numer to string, bit is width of number*/
74     std::string numtostr(int num,short bit);
75     /*string to number*/
76     int strtonum(std::string str);
77     /*get the table name*/
78     std::string GetTableName(std::string buffer,int start,int &rear);
79     /*get the address of the table and return the block information */
80     int GetTableAdress(std::string t_name,int &suitable_block);
81     /*get the size of the file*/
82     int fetchBlockNum(std::string t_name);
83     /*get table information*/
84     std::vector <std::string> tablelist;
85     int tablenum = 0;
86 };

```

成员变量

- `BM`: Buffer Manager的引用
- `TableList`: 用于记录已创建表的指针链表
- `tablenum`:用来记录现有的表的数量

成员函数

- `void CreateTable(std::string t_name, Attribute attribute, int primary, Index index)`

创建一个新表记录，并通过调用Buffer Manager为其分配一个内存page，输入变量位**表名、列名、主键位置、索引**

- `void DropTable(std::string t_name)`

根据输入的**表名**删掉对应的表的记录，并相应回收已分配的内存。

- `void CreateIndex(std::string t_name, std::string a_name, std::string i_name)`

创建一个新的索引，输入分别为**表名、列名、索引名**

- `std::string IndextoAttr(std::string t_name, std::string i_name)`

通过输入的**表名、索引名**搜索对应的一个列

- `void DropIndex(std::string t_name, std::string i_name)`

通过输入的**表名、索引名**删除对应的一个列

- `bool IsTable(std::string t_name)`

通过输入的**表名**判断是否已经有相同名字的表存在

- `bool IsAttribute(std::string t_name, std::string a_name)`

通过输入的**表名、列名**判断是否已经有相同名字的列存在

- `Attribute GetAttribute(std::string t_name)`

由输入的**表名**得到其中所有的列名

- `void PrintTable(std::string t_name)`

由输入的**表名**得到该表的所有

- `Index GetIndex(std::string t_name)`

由输入的**表名**的到该表的索引

- `std::vector <std::string> GetAllTable()`

返回已有的所有表名

- `std::string numtostr(int num, short bit)`

将数字转化成便于记录的字符串

- `int strtonum(std::string str)`

将记录中的字符串转化成为数字

- `std::string GetTableName(std::string buffer, int start, int &rear)`

通过传入的Buffer Manager和需要搜索的始末位置，从已开辟的内存Page中得到所有记录中的表名

4.4 Record Manager

Record Manager模块插入、删除、查询功能实现相关的代码封装在 `RecordManager` 类中，对应的代码文件是"RecordManager.h"和"RecordManager.cpp"。在 `RecordManager` 类中还定义了 `BufferManager`、`API`、`CatalogManager` 类的指针以便于调用其中的一些函数。为了判断记录是否符合条件定义了 `Condition` 类，其中提供了对记录进行判断的接口。

RecordManager 类

```
1  class RecordManager {
2  public:
3      BufferManager *bm;
4      CatalogManager *cm;
5      API *api;
6
7      string getTableFileName(string tablename);
8      string getIndexFileName(string tablename, string indexname);
9      //创建table时调用
10     //创建存储table对应的文件
11     //@tablename:table名称
12     int createTable(string tablename);
13     //删除table时调用
14     //output所有buffer中的block并删除磁盘文件
15     //@tablename:table名称
16     int dropTable(string tablename);
17     //创建index调用
18     //创建存储index的对应文件
19     //@indexname:index名称
20     int createIndex(string tableName, string indexname);
21     //删除index时调用
22     //output所有buffer中的block并删除磁盘文件
23     //@indexname:索引名称
24     int dropIndex(string tableName, string indexname);
25
26     //插入一条record
27     //将record写入buffer中的page并设置page为dirty
28     //@tablename:待插入的table名称
29     //@record: 指向待插入record的指针
30     //@recordSize:待插入record的大小
31     int recordInsert(string tablename, char* record, int recordSize);
32
33     //根据条件查找record并显示
34     //@tableName:表名
35     //@attributeNameVector:select后的attribute
36     //@conditionVector:where后的条件
37     int recordAllShow(string tableName, vector<string>*
attributeNameVector, vector<Condition>* conditionVector);
38
39     //找出符合条件的record条数并返回
40     //@tableName:表名
41     //@attributeNameVector:select后的attribute
42     //@conditionVector:where后的条件
43     int recordAllFind(string tableName, vector<Condition>*
conditionVector);
44     //删除符合要求的record, 返回值为删除的record条数
45     //@tableName:表名
46     //@conditionVector:where后的条件
47     int recordAllDelete(string tableName, vector<Condition>*
conditionVector);
48     //insert表中的所有index,返回insert的条数
49     //@tableName:表名
50     //@indexName:索引名
51     int indexRecordAllAlreadyInsert(string tableName, string indexName);
52 }
```



```

53 //删除一个block中符合要求的记录
54 //@tableName:表名
55 //@conditionVector:
56 int recordBlockDelete(string tableName, vector<Condition>*
conditionVector, int pageID);
57
58 //检查一条record是否符合conditionVector中的一系列条件
59 //如果全部符合条件,返回true,否则返回false
60 //@recordBegin:指向record记录开始部分的指针
61 //@recordSize:record的大小
62 //@attributeVector:指向一个包含record所有attribute的vector
63 //@conditionVector:指向一个包含所有condition的vector
64 bool recordConditionFit(char* recordBegin, int recordSize,
vector<SingleAttribute>* attributeVector, vector<Condition>*
conditionVector);
65 int recordBlockShow(string table_name, vector<string>*
attributeNameVector, vector<Condition>* conditionVector, int pageID);
66 private:
67
68 int recordBlockFind(string tableName, vector<Condition>*
conditionVector, int pageID);
69 int indexRecordBlockAlreadyInsert(string tableName, string indexName,
int blockID);
70
71 //检查一条record的某个attribute是否符合condition
72 //符合返回true,不符合返回false
73 //@content:指向record中某一attribute
74 //@type:attribute的类型, -1表示float,0表示int,其他数字表示string
75 //@condition:指向某一个Condition类,即某一个条件
76 bool contentConditionFit(char* content, int type, Condition*
condition);
77
78 void recordPrint(char* recordBegin, int recordSize,
vector<SingleAttribute>* attributeVector, vector<string>
*attributeNameVector);
79 void contentPrint(char* content, int type);
80
81 char* findFirstUsableBlock(string tablename);
82 int findContentBegin(string table_name, char* block_content, int
recordSize);
83 bool noRecord(string table_name, char* recordBegin);
84 };

```

成员变量

- `bm`: 指向 (全局共用的一个) Buffer Manager的指针
- `cm`: 指向 (全局共用的一个) Catalog Manager的指针
- `api`: 指向 (全局共用的一个) API的指针

成员函数

- `getTableFileName(string tablename)`
根据table名称得到table文件的路径并返回
- `getIndexFileName(string tablename, string indexname)`
根据index名称得到index文件的路径并返回
- `createTable(string tablename)`

创建表格时调用，在./database/record目录下生成对应的文件

- `dropTable(string tablename)`
删除table文件时调用，输出buffer pool中所有该table的记录文件的page并删除磁盘文件
- `createIndex(string tableName, string indexname)`
创建索引时调用，在./database/index下创建对应的储存索引的文件
- `dropIndex(string tableName, string indexname)`
删除index时调用，输出buffer pool中所有该index文件的page并删除磁盘文件
- `recordInsert(string tablename, char* record, int recordSize)`
给定表名、指向一条记录的指针、记录大小，将给定的记录插入表中
- `recordAllShow(string tableName, vector<string>* attributeNameVector, vector<Condition>* conditionVector)`
给定需要显示的属性名和查询条件，进行查询并输出结果
- `recordAllFind(string tableName, vector<Condition>* conditionVector)`
在指定的表中进行查找并返回符合条件的记录条数
- `int recordAllDelete(string tableName, vector<Condition>* conditionVector)`
删除表中符合条件的所有记录
- `indexRecordAllAlreadyInsert(string tableName, string indexName)`

查询表中已经插入的Index数目并返回

- `recordBlockDelete(string tableName, vector<Condition>* conditionVector, int pageID)`
删除指定block中符合条件的记录
- `recordConditionFit(char* recordBegin, int recordSize, vector<SingleAttribute>* attributeVector, vector<Condition>* conditionVector)`
判断一条record是否符合给定的条件
- `recordBlockShow(string table_name, vector<string>* attributeNameVector, vector<Condition>* conditionVector, int pageID)`
在指定的block中查询符合条件的记录并显示

4.5 Index Manager

Index Manager驱动模块的代码主要在"IndexManager.h"和"IndexManager.cpp"中，而B+树实现、索引文件的在磁盘的记录代码则在B+树模块文件"BpTree.h"中。Index Manager共通过3个类实现：`IndexManager`、`BpNode`、`BpTree`。

4.5.1 IndexManager 类

```
1  class IndexManager
2  {
3  private:
4      //传入的API指针
5      API* api_;
6      //定义类型：索引文件名 -> B+树
7      typedef std::map<std::string, BpTree<int, int>*> INTMAP;
8      //定义类型：索引文件名 -> B+树
9      typedef std::map<std::string, BpTree<float, int>*> FLOATMAP;
```

```

10 //定义类型：索引文件名 -> B+树
11 typedef std::map<std::string, BpTree<std::string, int>*> STRINGMAP;
12 //映射 索引文件名 -> int键B+树
13 INTMAP int_index_map_;
14 //映射 索引文件名 -> float键B+树
15 FLOATMAP float_index_map_;
16 //映射 索引文件名 -> string键B+树
17 STRINGMAP string_index_map_;
18
19
20 public:
21     //0表示数据类型为float
22     static const int TYPE_FLOAT = 0;
23     //-1表示数据类型为int
24     static const int TYPE_INT = -1;
25     //正数表示数据类型为字符串，且值为字符串长度 + 1；如34表示char(33)
26
27     //构造函数
28     //利用API间接调用catalog manager获取各表已有的index，在磁盘上找到对应的文件重构
B+树
29     IndexManager(API* api);
30
31     //初始化函数
32     void init();
33
34     //析构函数
35     //将已有的所有B+树写到磁盘上，释放内存
36     ~IndexManager();
37
38     //创建索引
39     //@file_name: 索引文件名（由record manager中的函数生成并传入）
40     //@type: 索引的类型
41     void createIndex(std::string file_name, int type);
42
43     //删除索引
44     //@file_name: 索引文件名（由record manager中的函数生成并传入）
45     //@type: 索引的类型
46     void dropIndex(std::string file_name, int type);
47
48     //单键值搜索
49     //@file_name: 索引文件名（由record manager中的函数生成并传入）
50     //@key: 待搜索的键值
51     //@type: 索引的类型
52     //return:
53     //找到的话返回记录所在blockID
54     //找不到key则返回-1
55     //没有该index返回-2
56     int searchIndex(std::string file_name, std::string key, int type);
57
58     //（不要求范围搜索，故暂时不实现）多键值范围搜索
59     // std::vector<int>* searchIndexRange();
60
61     //插入单条索引
62     //@file_name: 索引文件名（由record manager中的函数生成并传入）
63     //@key: 索引键值
64     //@blockID: 索引对应记录所在磁盘文件的block编号（注意，该处文件指磁盘中的记录文件，
而非索引文件）
65     //@type: 索引的类型

```

```

66     void insertIndex(std::string file_name, std::string key, int blockID,
67         int type);
68     //根据传入键值删除单条索引
69     //@file_name: 索引文件名（由record manager中的函数生成并传入）
70     //@key: 待删除的键值
71     //@type: 索引的类型
72     void deleteIndexByKey(std::string file_name, std::string key, int
73         type);
74 };

```

成员变量

- API类指针 `api_`：用于调用API模块中的相关函数 `API::allIndexInfoGet()`，以初始化所有磁盘上的索引至内存中（重构对应的B+树）
- 索引文件名至B+树的3个（不同类型键值的）`std::map`，包括 `int_index_map_`、`float_index_map_` 和 `string_index_map_`；B+树的两个模板参数类型 `<key, value>` 分别为 `<int, int>`、`<float, int>` 和 `<string, int>`，其中 `key` 参数类型表示键值的数据类型，而 `value` 参数类型表示该键值对应的记录所在记录文件的 `blockID`（即用记录在record file中的block编号来找到索引对应的记录所在位置）
- 静态常量 `TYPE_FLOAT = 0`、`TYPE_INT = -1`，对应用于表示键值为 `int` 和 `float` 的 `type` 数值

成员函数

- `IndexManager(API* api)`
`IndexManager` 的构造函数，传入一个API对象的指针用于初始化成员变量 `api_`
- `void init()`
利用API的相关内部函数，初始化本次启动数据库前在磁盘上创建并保存的所有索引至内存中
- `~IndexManager()`
`IndexManager` 的析构函数，将3个 `std::map` 中的B+树写至磁盘上的索引文件中予以保存，释放相关的内存
- `void createIndex(std::string file_name, int type)`
创建索引接口，根据传入的索引**文件名**和键值类型，在对应的 `std::map` 中新增映射关系，并创建一棵新的B+树用于保存索引文件
- `void dropIndex(std::string file_name, int type)`
删除索引接口，根据传入的索引**文件名**和键值类型，在对应的 `std::map` 中删除对应的映射关系，删除内存中用于保存索引文件的B+树并释放其内存
- `int searchIndex(std::string file_name, std::string key, int type)`
键值等值查找接口，根据传入的索引**文件名**、键值和键值类型，在对应的 `std::map` 中找到对应的映射关系，调用该B+树的等值搜索接口找到对应的 `blockID` 并返回
- `void insertIndex(std::string file_name, std::string key, int blockID, int type)`
键值插入接口，根据传入的索引**文件名**、键值、记录所在记录文件的block编号、键值类型，在对应的 `std::map` 中找到对应的映射关系，调用该B+树的插入键值接口
- `void deleteIndexByKey(std::string file_name, std::string key, int type)`
键值删除接口，根据传入的索引**文件名**、键值、键值类型，在对应的 `std::map` 中找到对应的映射关系，调用该B+树的删除键值接口

4.5.2 B+节点类

```
1  template <class Key, class Value>
2  class BpNode // B+树结点类
3  {
4  public:
5      friend class BpTree<Key, Value>;
6  public:
7      BpNode(int MAX_KEY, bool isLeaf);
8      ~BpNode();
9  private:
10     bool isLeaf; // 是否为叶节点
11     int key_count; // 目前储存的key数目
12     int MAX_KEY; // 至多储存的key数目
13     Key* keys; // key数组
14     Value* values; // value数组
15     BpNode<Key, Value>** ptrs; // 结点指针数组, 对于叶节点, ptrs[MAX_KEY]指向右兄弟
16 };
```

成员变量

- 友元类 `BpTree<Key, Value>`, 使得B+树类可以直接访问节点类的私有成员
- 节点是否为叶节点 `isLeaf`
- 该结点目前储存的键值数目 `key_count`
- 一个结点至多储存的键值数目 `MAX_KEY`
- 储存的键值数组 `keys`
- 储存的数据数组 `values`
- 结点指针数组 `ptrs`, 对于叶节点, `ptrs[MAX_KEY]` 指向其右兄弟

4.5.3 B+树类

```
1  template <class Key, class Value>
2  class BpTree
3  {
4  public:
5      Value* search(const Key& key) const;
6      bool insert(const Key& key, const Value& value);
7      bool delete_single(const Key& key);
8      BpTree(std::string file_name, int key_size, int MAX_KEY);
9      void write_back_to_disk_all();
10     BpNode<Key, Value>* get_minNode();
11     void read_from_disk_all();
12     ~BpTree();
13 private:
14     void split_node(BpNode<Key, Value>* parent, BpNode<Key, Value>* node,
15 const int pos);
16     bool insert(BpNode<Key, Value>* node, const Key& key, const Value&
17 value);
18     void delete_entry(BpNode<Key, Value>* node, const Key& key);
19     BpNode<Key, Value>* search_node(const Key& key);
20     int init_file(std::string file_path);
21     int get_block_num(std::string file_path);
22     void free_entry(BpNode<Key, Value>* root);
23 private:
24     BpNode<Key, Value>* root; // 根结点
```

```

23     int MAX_KEY; // 至多储存的key数目
24     std::string file_name; //对应的索引文件名
25     int key_size; //键的大小: int、float直接取sizeof()1; string类为字符串的长度
26 };

```

成员变量

- 根结点 `root`
- 其中一个结点至多储存的键值数目 `MAX_KEY`
- 对应的索引文件名 `file_name`
- 一个单位键的大小 `key_size`

私有成员函数

- `split_node()`
进行过满结点的分裂
- `bool insert(BpNode<Key, Value>* node, const Key& key, const Value& value)`
把对应的键值/数据对插入指定的结点 `node` 中
- `delete_single()`
递归式地释放B+树结点内存
- `search_node(const Key& key)`
如果输入的键值 `key` 存在, 则返回这一键值所在的B+结点; 否则, 返回存在于B+树中大于这一键值的最小值所在的结点
- `init_file()`
根据输入路径初始化文件
- `get_block_num()`
获取索引文件的块数

公开成员函数

- `Value* search(const Key& key)`
在B+树中查找对应键值, 返回指向对应数据的指针, 如果不存在, 返回 `nullptr`
- `bool insert()`
插入对应键值/数据对, 返回是否插入成功 (失败当且仅当该键值已经存在)
- `bool delete_single()`
根据输入的键值在B+树中删去对应的键值/数据对
- `search_node(const Key& key)`
如果输入的键值 `key` 存在, 则返回这一键值所在的B+结点; 否则, 返回存在于B+树中大于这一键值的最小值所在的结点
- `get_minNode()`
返回B+树底层最左边的结点 (即储存了最小键值的结点), 方便对全部数据的遍历
- `read_from_disk_all()`
从磁盘中读入B+结点, 构造B+树
- `write_back_to_disk_all()`
把B+树储存的内容全部写入磁盘

4.6 Buffer Manager

Buffer Manager通过两个类实现：

- Page
- BufferManager

其中，Page 用于表示缓冲池中的单个页，而 BufferManager 掌管整个缓冲池。

4.6.1 Page 类

```
1  class Page
2  {
3  private:
4      bool valid_;
5      int block_id_;
6      std::string file_name_;
7      char content_[PAGESIZE];
8      int pin_;
9      bool dirty_;
10     bool reference_;
11 public:
12     Page();
13     void initialize();
14     void setValid(bool valid);
15     bool getValid();
16     void setFileName(std::string n);
17     std::string getFileName();
18     void setBlockID(int block_id);
19     int getBlockID();
20     void setPin(int pin);
21     int getPin();
22     void setDirty(bool dirty);
23     bool getDirty();
24     void setRefer(bool reference);
25     bool getRefer();
26     char *getBuffer();
27 };
```

成员变量

- 页内存空间 content_[PAGESIZE]
- 保存的内容属于的文件名 file_name_
- 保存的内容在其属于的文件中的block id block_id_
- 钉住数目 pin_
- 引用数目 reference_
- 是否为脏块 dirty_
- 是否有效（即该Page的内容是否有意义） valid_

成员函数

- 构造、析构 Page();
- 初始化 void initialize()
- 设置/返回上述成员变量值的若干函数
 - 设置/返回文件名 setFileName/getFileName

- 设置/返回block id `setBlockID/getBlockID`
- 设置/返回钉住数目 `setPin/getPin`
- 设置/返回引用数目 `setRefer/getRefer`
- 设置/返回脏块标记 `setDirty/getDirty`
- 设置/返回dirty标记 `setDirty/getDirty`
- 返回页（内存空间）首地址 `getBuffer`

4.6.2 BufferManager 类

```

1  class BufferManager
2  {
3  public:
4      BufferManager()
5      {
6          initialize(MAXPAGEPOOLSIZE);
7      }
8      BufferManager(int page_num)
9      {
10         initialize(page_num);
11     }
12     ~BufferManager();
13     char* fetchPage(std::string file_name, int block_id);
14     int fetchPageID(std::string file_name, int block_id);
15     void markPageDirty(int page_id);
16     int pinPage(int page_id);
17     int unpinPage(int page_id);
18     int outputPage(int page_id);
19     void outputFile(std::string file_name);
20 private:
21     Page* page_pool_;
22     int page_pool_size_;
23     void initialize(int page_num);
24     int offerPageID();
25     int loadDiskBlock2Page(int page_id, std::string file_name, int
26     block_id);
27 };

```

成员变量

- 缓冲池 `page_pool_`
- 总页数 `page_pool_size_`
- 其余成员（如用于替换策略的变量）

成员函数

- 构造 `BufferManager()`：调用 `initialize()`
 - `initialize()`：初始化
- 析构 `~BufferManager()`：在程序结束时调用 `flushPage()` 将所有page写回磁盘
 - `loadDiskBlock2Page()`：将指定文件中的指定block载入指定的page
- `fetchPage()`：返回文件中的某一block在缓冲池中的page首地址；如果不在buffer pool则要载入后返回
 - `fetchPageID()`：找到文件中的某一block在缓冲池中的page id
 - `offerPageID()`：找到一个闲置的页；可能要实行时钟替换策略进行替换

- `loadDiskBlock2Page()`：将指定文件中的指定block载入指定的page
- `markPageDirty()`：标记page_id被修改了，即设为dirty
- `outputPage()`：将缓冲池中的指定页写到磁盘文件中的指定块
 - `loadDiskBlock2Page()`：将指定文件中的指定block载入指定的page
- `outputFile()`：将缓冲池中文件名为 `file_name` 的所有页写到磁盘上
- `pinPage()`、`unpinPage()`：钉住页面(pin_count++)、解除页面钉住状态(pin_count--)。防止页被替换！

五、界面说明

运行MiniSQL，初始界面如下：

```
-----MiniSQL interface-----
MiniSQL>
```

进入 `MiniSQL interface` 后，左边出现 `MiniSQL>` 提示输入信息

MiniSQL支持语句逐条输入，每条语句以分号结尾。如创建表格：

```
MiniSQL> create table student (
MiniSQL>             sno char (8),
MiniSQL>             sname char(16) unique,
MiniSQL>             sage int,
MiniSQL>             sgender char (1),
MiniSQL>             primary key ( sno )
MiniSQL> );
Create table student successfully!
(0.01 sec)
```

按下回车后，**MiniSQL**会给予反馈，（如上图的"Create table student successfully!"），并在最后一行打印该操作消耗的时间。同理，创建索引：

```
MiniSQL> on student ( sname );
Create index stunameidx on table student successfully!
(0.02 sec)
```

插入记录：

```
MiniSQL> insert into student
MiniSQL> values('111', 'A', 11, 'F'),
MiniSQL>          ('222', 'B', 22, 'M'),
MiniSQL>          ('333', 'C', 33, 'F'),
MiniSQL>          ('444', 'D', 44, 'M'),
MiniSQL>          ('555', 'E', 55, 'F'),
MiniSQL>          ('666', 'F', 66, 'M'),
MiniSQL>          ('777', 'G', 77, 'F');
insert result:#####
insert 7 record(s) on student successfully!
#####
(0.01 sec)
```

查询记录：

```

MiniSQL> select sno, sname, sage, sgender
MiniSQL> from student
MiniSQL> where sno <> '111';
select result: #####
sno      sname    sage    sgender
222      b        22      m
333      c        33      f
444      d        44      m
555      e        55      f
666      f        66      m
777      g        77      f
select 6 record(s) on student successfully!
#####
(0.02 sec)

```

当出现语句错误时，MiniSQL会报错

```

MiniSQL> BULLSHIT;
syntax error!

```

MiniSQL还支持文件读取输入。输入 `execfile [文件名]`，即可按顺序执行文件中的语句。

```

MiniSQL> execfile test.sql;
Create table student successfully!
insert result:#####
insert 7 record(s) on student successfully!
#####
select result: #####
sno      sname    sage    sgender
111      a        11      f
222      b        22      m
333      c        33      f
444      d        44      m
555      e        55      f
666      f        66      m
777      g        77      f
select 7 record(s) on student successfully!
#####
Drop table student successfully!
(0.05 sec)

```

输入 `quit;`，即可退出MiniSQL；此时程序弹出 `Goodbye!`，退出程序。

```

MiniSQL> quit;
Goodbye!

```

六、系统测试
