浙江大学

数据库系统实验报告

作业名称：	MiniSQL
姓　　名：	官泽隆
学　　号：	3180103008
电子邮箱：	学号@zju.edu.cn
联系电话：	18888910213
指导老师：	孙建伶

2020 年 6 月 18 日

BufferManager, RecordManager
详细设计报告

一、 与本报告有关的实验目的

1. 设计并实现一个精简型单用户 SQL 引擎(DBMS)MiniSQL. 实现表的建立/删除以及表记录的插入/删除/查找；帮助索引的建立/删除。
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、 与本报告有关的系统需求

1. 数据类型 要求支持三种基本数据类型：int，char(n)，float，其中 char(n)满足 1 <= n <= 255
2. 一个表最多可以定义 32 个属性，各属性可以指定是否为 unique
3. 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
4. 支持每次一条记录的插入操作；
5. 支持每次一条或多条记录的删除操作。(where 条件是范围时删除多条)

三、 实验环境

1. Windows 10 Professional
2. Microsoft Visual Studio 2017

四、 模块设计

1. Buffer Manager
   i. 功能描述：Buffer Manager 是一个全局单例。所有使用 Buffer Manager 进行文件数据管理的客户将共享一个由许多内存页组成的缓冲区。但对客户而言，缓冲区是透明的，Buffer Manager 就提供以页为单位的文件读写。具体而言，提供如下接口：

```
static BufferManager& instance() { return bm; }      // 获取全局单例
~BufferManager();                                    // 析构函数
const void* getPage_r(p_Page p);                 // 获取用于读的页面
void* getPage_w(p_Page p);                       // 获取用于写的页面
void set_dirty(p_Page p);              // 调用后，允许在原来只读的页面做写操作
unsigned totalPages(string filename);        // 获取某文件的总页面数
// int: 0 on success below
int pinPage(p_Page p);                 // 锁定缓冲区的页，不允许替换出去
void unpinPage(p_Page p);              // 解锁缓冲区的页，允许替换
int create_file(string filename);                // 创建文件
int delete_file(string filename);                // 删除文件
void flush();                                    // 将所有脏页写回磁盘
```

ii. 主要数据结构：

a) 接口上，使用(filename, pageNum)组成的元组 p_Page 来定位文件中的页

b) 使用**哈希表**来快速由 p_Page 映射到实际的内存页

```
unordered_map<p_Page, Buf_Page, MyHash> pages;
```

c) 使用**哈希表**维护文件的总页数

```
unordered_map<string, unsigned> _totalPages;
```

2. Record Manager

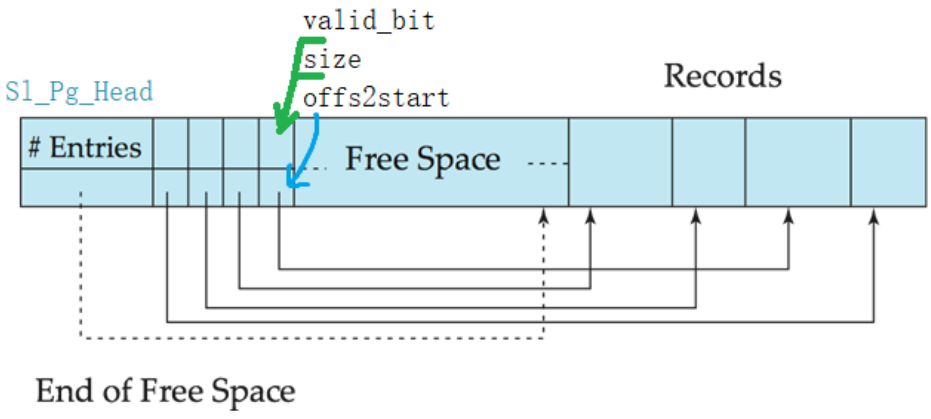
i. 功能描述：实现了设计指导的功能要求“管理记录表中数据的数据文件。主要功能为实现数据文件的创建与删除（由表的定义与删除引起）、记录的插入、删除与查找操作”。具体而言，提供如下接口：

```
int create_table(string tableName); // 0 for success
int drop_table(string tableName); // 0 for success
    // no candidates: full table scan; otherwise index range scan
vector<vector<string>> select(const string &tableName, const vector<Condition> &conds,
    const vector<p_Entry> &candidates = vector<p_Entry>{}); // 返回值是面向显示的
int insert(string tableName, std::vector<std::string> s_vals);    // 0 for success
int delete_rec(string tableName, vector<Condition> conds,
    const vector<p_Entry> &candidates = vector<p_Entry>{}); // 返回删除记录条数
void init_index(const string & indexName); // 向新 indx 提供已有键值对，以初始化
```

ii. 主要数据结构

a) 根据系统需求，把 C++对应的三种基本数据类型（int,double,std::string）做了一个 **wrapper** 以适配，提炼一个**抽象基类（ABC）**出来，以便用统一的代码读写各个字段，提高代码可读可维护性。

b) **分槽页（Slotted Page Structure）**，于课本 ch10 描述过，如下图

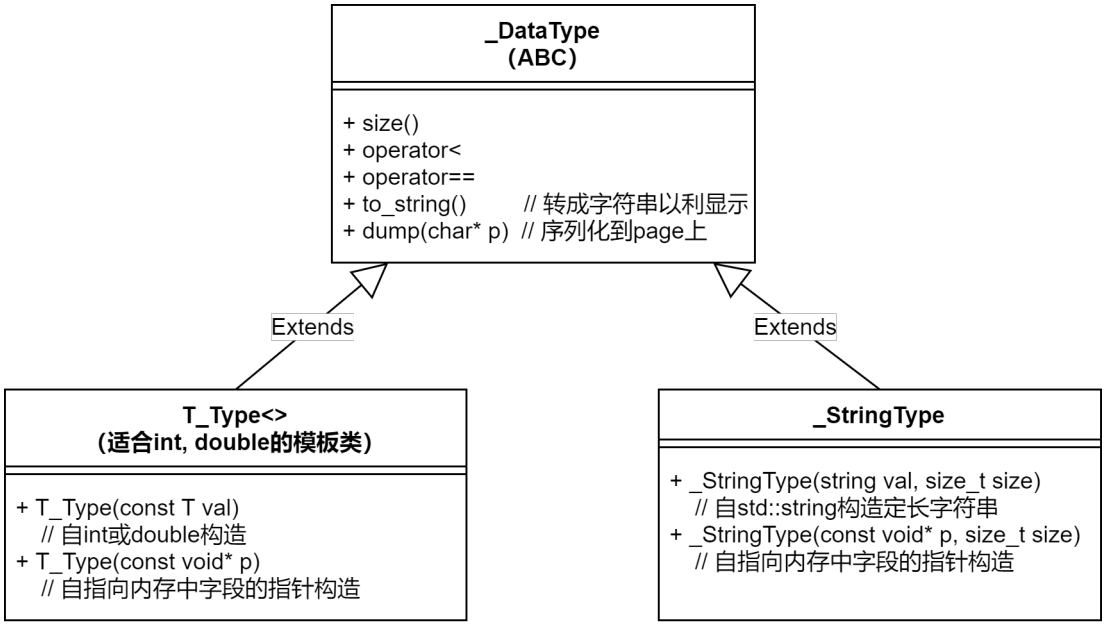


对应 C++的结构声明：

```
struct Sl_Pg_Head
{
    unsigned n_entries;
    unsigned end_fspace; // 页内的free space是[第n_entries项 Entry, end_fspace)
    Entry ent;
};
struct Entry
{
    unsigned offs2start; // relative to the start of page
    unsigned short size;
    bool valid;
};
```

c) 接口上使用(pageNum, offs)构成的元组 p_Entry 来定位表的 Entry.

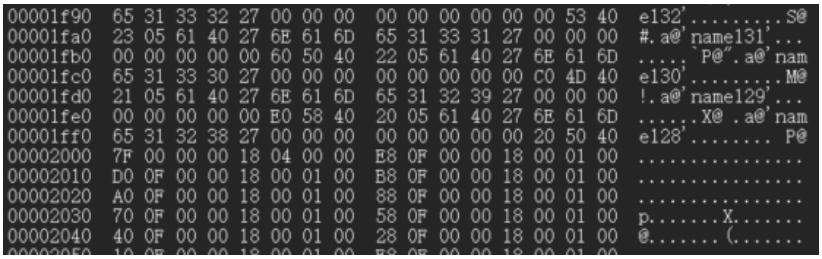
iii. ii.a) 提到的 UML 类图



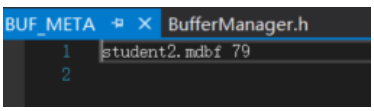
3. Relevant DB Files

i. 存储文件格式说明

a) tableName.mdbf 以二进制形式存储 tableName 表中的记录数据。一瞥



b) BUF_META 以文本文件的形式存储各.mdbf 文件的总页数



五、 模块实现

1. Buffer Manager
   a) .cpp 实现文件中，对于每个页，除了指向实际内存位置的 `void*` 指针外，Buffer Manager 还簿记该页：是否 `dirty`；是否 `pinned`；以及用于 clock 替换算法的 reference bit. 每个页的这些信息封装到 `struct Buf_Page` 中。
   ii. Buffer Manager 维护一个由许多内存页组成的缓冲区

Algorithm 1 getPage

```
Require: 用于定位该页的指针 p
  if 该页不在内存中 then
    打开文件 // 从磁盘中读入该页到内存
    if 如果页号对应的偏移超出当前文件大小 then
      文件大小膨胀为两倍偏移 // 类似 std::vector 的 capacity 策略
    end if
    根据访问的页面，更新文件总页数 // 文件总页数好比 size，capacity 和 size 不一样
    alloc_space() 分配内存空间，得到一个 void* 指针
    二进制 IO 方法 fin.read(char*, unsigned) 把页从磁盘读入内存
    哈希表中构造 Buf_Page 结构
  end if
  置该页 reference bit
  根据读还是写设置脏 bit
  return 哈希表中 p 对应 Buf_Page 的 void* 内存指针
```

Algorithm 2 alloc_space

```
  if buffer 未满 then
    直接 new
  else
    while 未寻到替换页面 do
      for all 哈希表中 Buf_Page do
        if 置了 clk_ref then
          清除 clk_ref
        else
          if 该页脏 then
            该页写回磁盘
          end if
          窃取该页 void* 内存指针
          哈希表中删除该页记录
        end if
      end for
    end while
  end if
  return 之前得到的 void* 内存指针
```

Algorithm 3 Buffer Manager 析构

```
for all 哈希表中 Buf_Page do
  if 该页脏 then
    该页写回磁盘
  end if
end for
for all 哈希表中 Buf_Page do
  释放 Buf_Page 先前用 new 申请的内存空间
end for
打开 BUF_META 文件
把内存中 _totalPages 哈希表的信息以文本形式写出
return
```

2. Record Manager

i. 四、2.ii.a)提到的 **wrapper**，实现上是非常简单的，举例如下

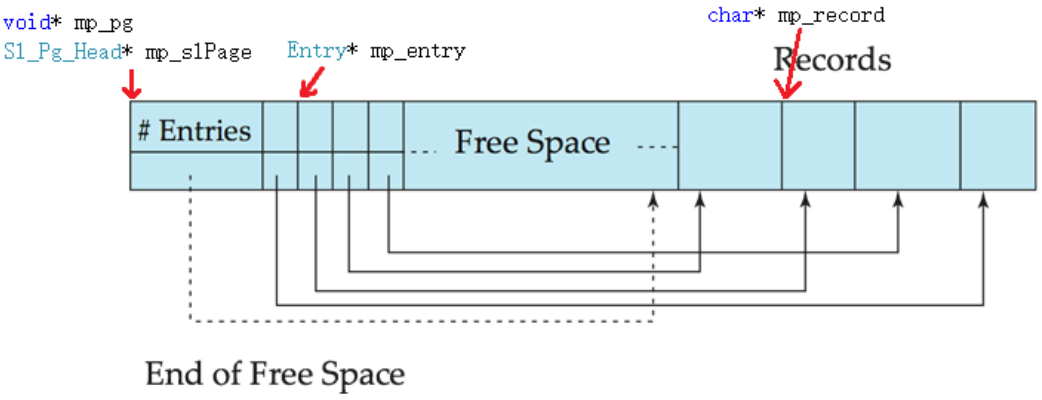
```
template<typename T> // int, double
class T_Type : public _DataType
{
    T d;
public:
    T_Type(const T val) : d(val) {}
    T_Type(const void* p) { d = *(T*)p; }
    size_t size() const { return sizeof(T); }
    bool operator<(const _DataType &d2) const { return d < dynamic_cast<const T_Type &>(d2).d; }
    bool operator==(const _DataType &d2) const { return d == dynamic_cast<const T_Type &>(d2).d; }
    string to_string() const { return std::to_string(d); }
    void dump(char* p) const { *(T*)p = d; }
};
```

ii. 对数据文件的读写

我的总结是，这其实是一个**根据提示进行序列化和反序列化**的任务。借助 Buffer Manager 提供的 `const void* getPage_r(p_Page p); void* getPage_w(p_Page p);`接口，任务转化为对一块**内存**的读写。RM 通过分槽页和 Catalog 提供的信息完成这一任务。

a) 分槽页的遍历

如下图，省略了可能的 const 修饰符。注意：这些指针都是指向**内存位置**的指针。



RM 首先得到 `void* mp_pg`，然后重新解释为 `Sl_Pg_Head* mp_slPage`.

由此得到 `mp_slPage->n_entries`，并计算出指向第一个 Entry 的 `Entry* mp_entry;`

对于每个 `Entry* mp_entry`，可以由`(char*)mp_pg + mp_entry->offs2start` 算出 `char * mp_record`.

b) 读（**反序列化**）

一个记录中的各个字段紧邻存放。顺序由 Catalog 提供的 vector<Column>确定；每个字段对应的数据类型由 Catalog 提供的 Column.type 确定。对某个**记录**的读取过程如下：

```
for (size_t i = 0; i < cm.getTableInfo(tableName).metadata.size(); i++) // get each field
{
    auto col = cm.getTableInfo(tableName).metadata[i];
    pair<DataType, int> type = eType(col.type);
    _DataType *data = mk_obj(type, mp_record);
    vs_record.push_back(data->to_string());
    mp_record = (const char*)mp_record + data->size(); // step to next field
    delete data;
}
```

对某个**字段**的读取过程提取为如下函数：

```
_DataType* RecordManager::mk_obj(std::pair<DataType, int> &type, const void * mp_record)
{
    switch (type.first)
    {
    case IntType:
        return new T_Type<int>(mp_record);
        break;
    case FloatType:
        return new T_Type<double>(mp_record);
        break;
    case StringType:
        return new _StringType(mp_record, type.second);
        break;
    default:
        return nullptr;
        break;
    }
}
```

c) 写（**序列化**）

对某个**字段**的序列化过程就是_DataType 的 dump()方法。这是一个虚方法，因此只要你有基类指针数组，就可以方便地把一个**记录**中的各个字段相继写出，如下：

```
void RecordManager::dump_rec(char * mp_record, const _DataType * const dataArr[], unsigned n)
{
    for (size_t i = 0; i < n; i++)
    {
        dataArr[i]->dump(mp_record);
        mp_record += dataArr[i]->size();
    }
}
```

iii. vector<vector<string>> select(tableName, conds, candidates);

分为两个阶段：找名单，转成 vector<vector<string>>返回。有 index 的情况下，搜寻范围缩小为 candidates 的成员。

```
vector<vector<string>> RecordManager::select(const string & tableName, const vector<Condition>& conds, const vector<p_Entry>& candidates)
{
    if (candidates.empty()) return to_print(tableName, full_table_scan(tableName, conds));
    else return to_print(tableName, range_scan(tableName, conds, candidates));
}
```

a) vector<p_Entry> full_table_scan(tableName, conds);

一张表的全部数据都存于 tableName.mdbf 中，因此算法是简单的：

```
traverse each page:
    traverse each record in the page:
        check conds_fit.
```

b) conds_fit

上层决定使用 string 来传递条件的值，因此我又写了一个类似 ii.b) mk_obj()的小函数由 string 转_DataType.

```
_DataType* RecordManager::mk_obj(const pair<DataType, int>& type, const string & val)
{
    switch (type.first)
    {
    case IntType:
        return new T_Type<int>(stoi(val));
        break;
    case FloatType:
        return new T_Type<double>(stod(val));
        break;
    case StringType:
        return new _StringType(val, type.second);
        break;
    default:
        return nullptr;
        break;
    }
}
```

在 ii.b)的读每个字段的过程中，我们都看一眼 conds，如果有涉及到这个字段的条件，我们就用_DataType 提供的关系运算符作比较，检查是否符合条件。

c) cond_fit

```
bool RecordManager::cond_fit(const Condition & c, const _DataType *data, const _DataType * cond_val)
{
    bool fit = true;
    switch (c.op)
    {
    case opType::E:
        if (!(*data == *cond_val)) fit = false;
        break;
    case opType::NE:
        if (*data == *cond_val) fit = false;
        break;
    case opType::L:
        if (!(*data < *cond_val)) fit = false;
        break;
    case opType::GE:
        if (*data < *cond_val) fit = false;
        break;
    case opType::G:
        if (*data < *cond_val || *data == *cond_val) fit = false;
        break;
    case opType::LE:
        if (!(*data < *cond_val || *data == *cond_val)) fit = false;
        break;
    default:
        break;
    }
    return fit;
}
```

iv. int delete_rec(tableName, conds, candidates);

是简单的懒惰删除，这也是为什么我们的实现较课本 ch10 多了个 valid_bit. 先通过与 select()一样的策略确定名单（因此只需读权限）；完成后对 vector<p_Entry> list 中的各成员重新读取页面（写权限），置 mp_entry->valid = false;

v. `int insert(tableName, vector<string> s_vals);`

首先做 integrity check:（目前只有 unique 键）会从 Catalog 读取表的字段信息，遇到 unique 的 column，就会在 conds 中加一条(col==val)的条件。最后通过 `select(tableName, conds)` 的返回值判断 unique 键值是否已经存在。

懒惰删除的代价是插入的复杂性增加了。有两种策略进行插入：

Algorithm 4 快速插入

打开最后一页（当心总页数为 0 的边界情况，这里略去）
if 该页 free_space 不够容纳 record 和 entry **then**
  新一页，init
end if
在最后一个 entry 后写入新 entry 的信息
从 $end\_fspace - rec\_size$ 起写 record 到页上
更新该页 header
return

Algorithm 5 插入懒惰删除导致的空穴

for all 表中的页 **do**
  for all 页中的 Entry **do**
    if 该 Entry 被懒惰删除且记录大小足够 **then**
      到该 Entry 对应的 mp_record 处，写要插入的 record 到页上
      该 Entry 重新置为 valid
      return
    end if
  end for
end for
快速插入（未找到懒惰删除导致的空穴）

显然，后一种策略的代价很高。我们通过控制两个策略执行的频率来摊还由懒惰删除引起的插入代价。暂定每 `const unsigned FILL = 0x100` 次执行一次第二种策略。

六、 遇到的问题及解决方法

1. Dual paging 与标准 IO 流
   a) Buffer Manager 的实现不能避免上课讲的 Dual paging 问题
   b) 队友反映 Buffer Manager 提供的接口受到分页限制，比较难用，没有标准 IO 流的那种流畅感觉。但是如果要解决，相当于要把标准 IO 流的轮子再造一遍，很不现实。
   c) 其实标准 IO 流的底层也有一个 buffer（它的实现肯定更精致，不知能否避免 Dual paging 问题），要是能够对它的工作方式进行微调，使得它能够以页的粒度进行磁盘 IO，那么既适应了 DBMS 的效率需求，用户程序员也完全感受不到分页的限制，那该多好啊。遗憾的是，我至今还没有找到门道。
2. 由于整个程序串行，最后也没有使用 pinPage 有关功能。这可以作为提高效率的小优化。

3. Record Manager：其实我从一开始就一直在想：如何用固定的代码处理运行阶段才能确定的数据库 schema？据队友说，Java 等语言中的“反射”机制能够比较优雅地应对这种需求，但是自己并不会这些高级语言。最后复习 C++ OOP arsenal，决定运用多态机制作为权宜之计。
4. RM 的操作其实还是比较危险，因为数据结构是程序直接读写内存而形成的，除了自己小心编程、多做测试和探索性调试外，没有外界的监督信息来避免错误。

七、 总结

1. 首次自己动手实现了硬盘上的数据结构。尽管分槽页（Slotted Page Structure）是非常简单的记录存储结构，可是真的让我实现，倒还是花了不少功夫。“纸上得来终觉浅，绝知此事要躬行。”
2. Buffer Manager 还是差点意思。不知这种数据库底层、硬盘读写的需求有没有优雅的解决方案
3. 一些公用接口、API 没有事前商量好。比如沟通字段的数据类型，有的用 string，有的用 enum，导致一些代码用于互相转换，意义不大