个人设计报告

——彭子帆

BufferManager模块：

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. 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

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，块的大小为文件系统与磁盘交互单位的整数倍，定为4KB。

BufferManager由两个类构成，外部接口如下：

```
1.  struct blockNode
2.  {
3.      int offsetNum; // the offset number in the block list
4.      bool pin;  // the flag that this block is locked
5.      bool ifbottom; // flag that this is the end of the file node
6.      char* fileName; // the file which the block node belongs to
7.      friend class BufferManager;
8.
9.  private:
10.     char *address; // the content address
11.     blockNode * preBlock;
12.     blockNode * nextBlock;
13.     bool reference; // the LRU replacement flag
14.     bool dirty; // the flag that this block is dirty, which needs to written back to the disk later
15.     size_t using_size; // the byte size that the block have used. The total size of the block is BLOCK
    _SIZE . This value is stored in the block head.
16.
17. };
18.
19. struct fileNode
20. {
21.     char *fileName;
22.     bool pin; // the flag that this file is locked
23.     blockNode *blockHead;
24.     fileNode * nextFile;
25.     fileNode * preFile;
26. };
```

> **Page**类为其他组件与Buffer进行数据交换的媒介，其他组件通过设定page中的tableName，attributeName，pageType，PageIndex 传 递 给 BufferManager，BufferManager通过page中的信息从硬盘或者从缓冲区获取数据或者写入数据。需要 写入和读取的数据都在char数组pageData中。

```
class BufferManager
{
private:
    fileNode *fileHead;
    fileNode file_pool[MAX_FILE_NUM];
    blockNode block_pool[MAX_BLOCK_NUM];
    int total_block; // the number of block that have been used, which means the block is in the list.
    int total_file; // the number of file that have been used, which means the file is in the list.
    void init_block(blockNode & block);
    void init_file(fileNode & file);
    blockNode* getBlock(fileNode * file, blockNode* position, bool if_pin = false);
    void writtenBackToDiskAll();
    void writtenBackToDisk(const char* fileName, blockNode* block);
    void clean_dirty(blockNode &block);
    size_t getUsingSize(blockNode* block);
    static const int BLOCK_SIZE = 4096;

public:
    BufferManager();
    ~BufferManager();
    void delete_fileNode(const char * fileName);
    fileNode* getFile(const char* fileName, bool if_pin = false);
    void set_dirty(blockNode & block);
    void set_pin(blockNode & block, bool pin);
    void set_pin(fileNode & file, bool pin);
    void set_usingSize(blockNode & block, size_t usage);
    size_t get_usingSize(blockNode & block);
    char* get_content(blockNode& block);
    static int getBlockSize() //Get the size of the block that others can use.Others cannot use the block head
    {
        return BLOCK_SIZE - sizeof(size_t);
    }


    blockNode* getNextBlock(fileNode * file, blockNode* block);
    blockNode* getBlockHead(fileNode* file);
    blockNode* getBlockByOffset(fileNode* file, int offestNumber);

};
```

> 对于文件的操作使用了POSIX。为方便起见，所有的文件打开操作全部由BufferManager自动提供。当用户需要写入特定文件的时候BufferManager会自动检查 文件的存在或是否打开并获取文件句柄。文件句柄使用了map进行存储，其索引值为相应的表名索引名。缓存使用了一个Page数组进行内存缓存，为了实现LRU使用了一个与缓冲区大小相同的计数器来记录缓冲块没有被用的时间，为了实现替换时 写回使用了一个与缓冲区大小相同的bool数组记录缓冲区是否被写过，为了实现pin 使用了一个bool数组来记录pin过的缓冲区。

下面介绍读写的具体实现

readPage()

首先判断出缓冲区内是否有需要的页，如果有，读取缓冲区，改变计数器，将除了当前读取的缓冲页的其他缓冲页的计数器全部加1，返回，如果没有,检查当前文件是否被打开，没打开通过系统调用获取文件句柄，打开了从map中获取文件句柄， 通过文件句柄用系统调用直接读取文件的页，读取完了以后寻找当前缓冲区中计数器最大且没有pin的页，进行替换，如果替换的页dirty需要写回。

writePage()

将read中的操作改变成write即可。

allocatePage()， deallocatePage()

在一个文件内部回收的页由0号页作为头使用页内头四个字节作为指针进行连接。

allocatePage()会分配给参数Page页号作为新的页，分配时如果回收链表有页，则从 链表获取页，如果链表为空即0号页指向-1，计算文件大小/PAGESIZE的到文件尾部 页号。删除页将页插入链表中即可。

测试方法：

通过调用allocate和deallocate输出页号与预期相同。

一开始不经过缓冲区跑整个系统成功后加入缓冲区发现功能正常。 由于时间有限没有在Buffer上另外做过多测试。

Record Manager模块

Record Manager负责处理表的数据的插入、查找、删除操作。Record Manager模块从API模块、Catalog模块处接收信息，并利用Buffer Manager所提供的方法对数据在buffer中进行筛选和操作。可以说，Record Manager模块是与各个模块联系都很紧密的一个模块。其具体实现如下类：

```
1.  class Record{
2.  public:
3.      Record(){}
4.      BufferManager bm;
5.      API *api;
6.
7.      int tableCreate(string tableName);
8.      int tableDrop(string tableName);
9.
10.     int indexDrop(string indexName);
11.     int indexCreate(string indexName);
12.
13.     int recordInsert(string tableName, char* record, int recordSize);
14.
15.     int recordAllShow(string tableName, vector<string>* attributeNameVector, vector<Condition>* conditionVector);
16.     int recordBlockShow(string tableName, vector<string>* attributeNameVector, vector<Condition>* conditionVecto
    r, int blockOffset);
17.
18.     int recordAllFind(string tableName, vector<Condition>* conditionVector);
19.
20.     int recordAllDelete(string tableName, vector<Condition>* conditionVector);
21.     int recordBlockDelete(string tableName,  vector<Condition>* conditionVector, int blockOffset);
22.
23.     int indexRecordAllAlreadyInsert(string tableName,string indexName);
24.
25.     string tableFileNameGet(string tableName);
26.     string indexFileNameGet(string indexName);
27. private:
28.     int recordBlockShow(string tableName, vector<string>* attributeNameVector, vector<Condition>* conditionVecto
    r, blockNode* block);
29.     int recordBlockFind(string tableName, vector<Condition>* conditionVector, blockNode* block);
30.     int recordBlockDelete(string tableName,  vector<Condition>* conditionVector, blockNode* block);
31.     int indexRecordBlockAlreadyInsert(string tableName,string indexName, blockNode* block);
32.
33.     bool recordConditionFit(char* recordBegin,int recordSize, vector<Attribute>* attributeVector,vector<Conditio
    n>* conditionVector);
34.     void recordPrint(char* recordBegin, int recordSize, vector<Attribute>* attributeVector, vector<string> *attr
    ibuteNameVector);
35.     bool contentConditionFit(char* content, int type, Condition* condition);
36.     void contentPrint(char * content, int type);
```

-对于数据的三大操作，RecordManager实际的实现分别如下：

> **INSERT**：
> Insert into (table)
>
> > 用表名创建一个Table的类，然后通过**insertTuple**来插入这个属性值（并不判断属性正确）

> **SELECT**：
>
> > 1. select * / (attributes) from (table)
> >   用表名创建一个Table的类，然后通过**getAll**来读取这个属性
> > 2. select * / (attributes) from (table) where (conditions)
> >   用表名创建一个Table的类，然后通过**scan**的方法来获取所有符合条件的页号
> >   其中，Conditions可以为多组，实现方法是将多组返回的vector一

> **DELETE**：
>
> > 1. delete from (table) where (conditions)
> >   具体查找操作同SELECT中的第2条实现
> >   其中，Conditions可以为多组，实现方法是将多组返回的vector用
> >   O（n）的时间merge在一起
> >   最后，调用**deteleTuple**来删除这些数据
> > 2. delete from [table]
> >   用表名创建一个Table的类，然后通过**getAll**来读取所有的页号，再调

- 可见，Record Manager在数据库结构中，扮演的是次级任务实现者的角色。之所以这么说，是因为Record Manager的实际工作，在大多数情况下并不是直接对底层的操作，而是借助Buffer Manager和Index Manager的功能来实现数据库的操作，其中buffer则是重中之重，因为所有数据都是通过它经行实际交互的。当然，Record Manager也会需要将得到的数据经行处理，以便向上传递。

测试样例

表中空的时候先测试所有的函数，无问题。

- `PageIndexType insertTuple(vector< Attribute >);`
  尝试连续插入1000条记录，插入成功
- `void deleteTuple(PageIndexType);`
  尝试删除其中200条记录，插入成功
- `vector< PageIndexType > scan…(int , Attribute);`
  尝试多次选出其中的记录，选取成功
- `vector< PageIndexType > getAll();`
  尝试拿出所有记录，选取成功
- `vector< Attribute > getTupleAtPage(PageIndexType);`
  尝试拿出一个属性所有值提供给索引，选取成功
- `void printinfo(PageIndexType);`
  每次选取后使用输出结果，输出成功