存储以及缓存管理器

本实验主要涉及:

> - Buffer以及Frame(内存页)大小设置
> - Buffer以及Frame存储结构
> - 页格式
> - 页存储结构
> - 文件格式
> - 缓存技术
> - 哈希技术
> - 文件存储结构
> - 磁盘的接口功能以及缓存模块

Buffer(缓存)以及Frame(内存页)

Buffer以及Frame大小设置

Buffer一般指主存空间(内存),是CPU能直接访问的内容,Buffer可以视为frame组成的一维数组,当某页被需求时,将其加载到内存中,主流数据库中内存页(frame)与硬盘页(page)保证相同大小(防止碎片化),本项目中同样采用相同的策略,并且**我们规定内存大小为1024**

Buffer以及Frame存储结构

对于一个Frame,我们可以定义为一串连续的内存空间,其代码表示如下:

```
#define FRAMESIZE 4096
struct bFrame
{
    char field [FRAMESIZE];
};
```

由于Buffer由若干Frame组成,因此我们类似的可以定义:

```
#define DEFBUFSIZE 1024
bFrame buf[DEFBUFSIZE]; // or the size that the user defined by the input
parameter
```

上述为Buffer的物理表示,其可以被Buffer manager/文件系统或其他方式访问,提供所需页(page)

页格式

本次实验对于页属性仅保留$page_id$以及$page_size$,因此忽略页格式的影响

文件格式

本次实验建议使用基于目录的方式建立数据库文件,每个文件由一个索引页(目录)用于**顺序**储存所有其他页的指针,对应的数据页不储存任何文件指针,仅保留数据,额外要求,对于索引页的访问**必须是顺序**的(不能下标直接访问,相当于使用链表方式存储(存疑?)) 使用基于目录的方式主要出于可以快速获得对应的页,而不用对一长串的页进行遍历(即只需要对目录页扫一遍就能找到了,但是如果使用链表的方式,则需要对中间所有页进行一次遍历)

缓存技术

我们使用LRU作为实验涉及的仅有的替换策略,LRU每次总是找到最近最久未使用的元素进行替换,即对于LRU队列中按最后一次访问时间排序(升序,队首弹出,队尾压入)
使用LRU的原因处于这样一个假设,最近被使用的元素,在近期更有可能被访问,即时间局部性,并且其运行时间稳定

哈希技术

对于在内存中的页,使用一个buffer control block(BCB)来进行管理,一个BCB包括下面的信息:

> - page_id:标记对应物理页id,并且用于hash到对应的桶中进行管理
> - frame_id:标记对应内存页id,
> - page_latch:锁
> - fix_count:用于表示当前有多少个线程正在读它,作用相当于pin-count,当其值为0时才可以被释放
> - dirty_bit:脏标记用于判断是否需要写回

本次实验建议使用简单静态哈希,对于静态hash桶的容量时固定的,当满的时候使用溢出块来储存多余的数据,**对于单个桶的访问也是顺序的**,对于静态hash函数的选择可以参考下面的方式:

```
H(k)=(page_id)%buffer_size
```

对于BCB我们可以考虑使用下面的结构体来表示:

```
struct BCB
{
    BCB();          //构造函数
    int page_id;    //物理页id
    int frame_id;   //内存页id
    int latch;      //是否加锁
    int count;      //是否占用
    int dirty;      //是否写回
    BCB * next;     //链表形式
};
```

文件存储

本次实验中我们所有数据以及目录结构都存在一个文件中,并且我们对于该文件命名为**data.dbf**

类设计

数据存储管理器(Data Storage Manager)

```
class DSMgr
{
public:
    DSMgr();     //构造函数
    int OpenFile(string filename);  //打开指定数据文件,返回错误码
    int CloseFile();                //在关闭数据库时调用
    bFrame ReadPage(int page_id);   //从磁盘中读取某页到内存中,被FixPage调用,使用fseek,以及fread获取指定内容
    int WritePage(int frame_id, bFrame frm);     //将内存中某页写回到磁盘,在任意写回时刻被调用,返回值表示写了多少字节,使用fssek以及fwrite进行定向书写
    int Seek(int offset, int pos);  //将文件指针移动到指定位置
    FILE * GetFile();               //返回当前文件指针
    void IncNumPages();             //计数当前使用Page页+1
    int GetNumPages();              //获得当前内存中页数目
    void SetUse(int index, int use_bit);     //使用位图来表示当前页是否被占用,
    int GetUse(int index);                   //获取位图
private:
    FILE *currFile;                          //文件指针用于表示当前文件
    int numPages;                            //当前使用页数目
    int pages[MAXPAGES];                     //物理页
};
```

缓存管理器(Buffer Manager)

```
class BMgr
{
public:
    BMgr();     //构造函数
    // Interface functions
    int FixPage(int page_id, int prot); //查找给定page_id是否存在内存中,如果不在则选择某页替换,并返回对应frame_id
    Newpage FixNewPage();                    //申请一个空页,因此需要分别提供空闲的frame以及page,因此将<frame_id,page_id>作为返回值
    int UnfixPage(int page_id);          //接触对应page_id的一次占用,当对应占用为0时可以将其进行替换或者其他操作
    int NumFreeFrames();                 //返回当前内存中还空闲的页个数
    // Internal Functions
    int SelectVictim();                  //返回一个可以替换的页
    int Hash(int page_id);               //实现将page_id到frame_id的映射
    void RemoveBCB(BCB * ptr, int page_id); //将page_id对应的BCB删除
    void RemoveLRUEle(int frid);             //在LRU列表中移除对应frid
    void SetDirty(int frame_id);             //将对应frame_id脏标记置1,1表示需写回,0表示不用写回
    void UnsetDirty(int frame_id);           //将对应frame_id脏标记置0
    void WriteDirtys();                      //当系统结束运行时,对于脏标记为1的frame需要调用,保证数据一致性
    PrintFrame(int frame_id);                //打印对应帧消息,用于调试(?)
private:
    // Hash Table
    int ftop[DEFBUFSIZE];                    //frame到page的hash映射
```

```
    BCB* ptof[DEFBUFSIZE];                  //page到frame的映射
};
```