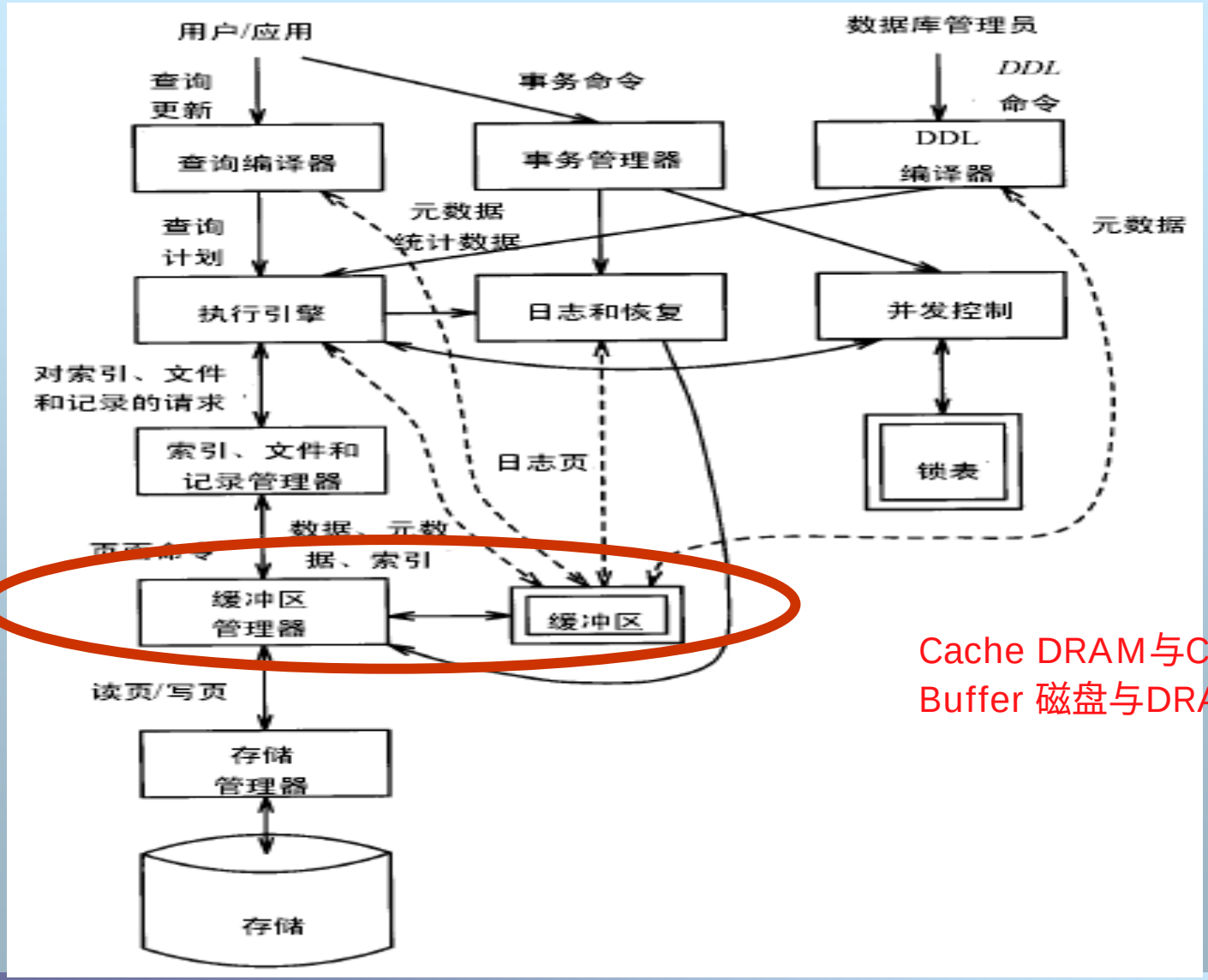
Buffer Management

金培权（jpq@ustc.edu.cn）

Idea: Minimize the count of disk I/Os by keeping likely-to-be-requested pages in memory (buffer frames)



Cache DRAM与CPU的高速缓存
Buffer 磁盘与DRAM的缓存

主要内容

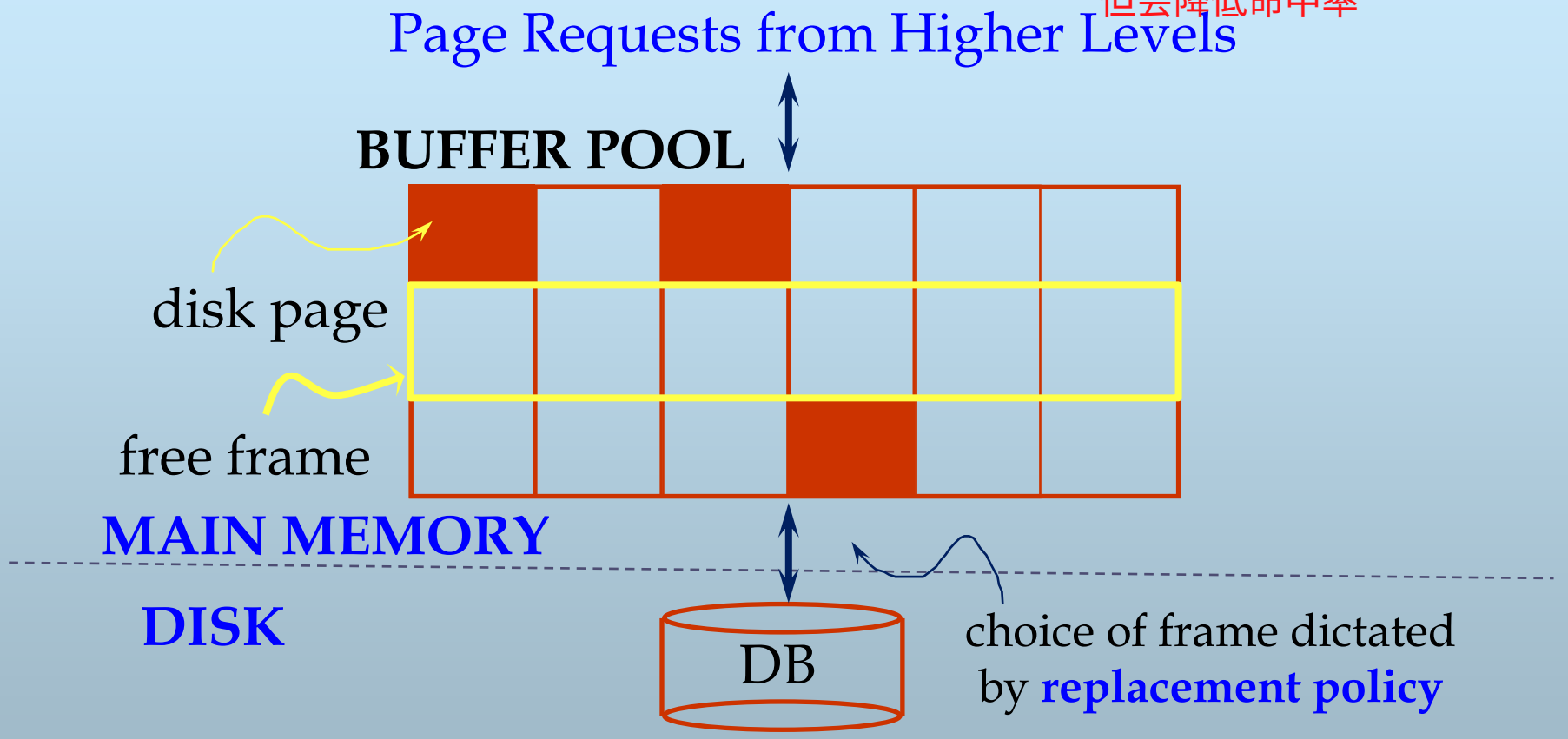
- 缓冲区结构
- 缓冲区置换算法
- 缓冲区管理的实现

buffer 通常使用数组实现，不要使用指针，数组每个元素大小设为64B，有利于CPU访问数据（CacheLine为64B）

一维数组实现，使用位图记录使用的page，维护LRU链表确定换出的page，空间不足时换出一个page

二维数组实现，一行作为一个frame，空间不足时换出一行（即一个frame）但会降低命中率

一、缓冲区结构



- ***Data must be in RAM for DBMS to operate on it!***
- ***Buffer Mgr hides the fact that not all data is in RAM***

1、frame的参数

- **Dirty**
  - **Frame中的块是否已经被修改**
- **Pin-count**
  - **Frame的块的已经被请求并且还未释放的计数，即当前的用户数**
- ***Others**
  - **Latch: 是否加锁**

如果换出的page的dirty为1，则需要将数据写回磁盘

每次换出可选择clear的page，但可能会需要遍历整个LRU链表，也可能使命中率下降

采用window参数，从LRU链表中的前w个page中选择换出的page（w设为LRU链表长度的二分之一）

每请求一个page时，该page的pin-count++

使用完释放后，该page的pin-count--

在并发的情况下会使用到

加在缓冲区的轻量锁

2、当请求块时

- If the requested block is not in the pool:
  - Choose a frame for replacement
  - If the frame is *dirty (some blocks are modified and haven't been written to the disk)*, write it to the disk
  - Read the requested block into the chosen frame
- *Pin (increment the pin-count* of the frame*)* the block and return its address.

2、当释放块时

- Requestor must *unpin* the frame containing the block
- Requestor must indicate whether block has been modified:
  - *dirty* bit is used for this.

二、缓冲区替换策略

- **Frame is chosen for replacement by a *replacement policy:***
  - **Least-recently-used (LRU), Clock, FIFO, MRU (Most-recently-used) etc.**
- **Only frames whose pin-count=0 are candidates**
- **Policy can have big impact on # of I/O's; depends on the *access*** 访问模式 ***pattern*.**

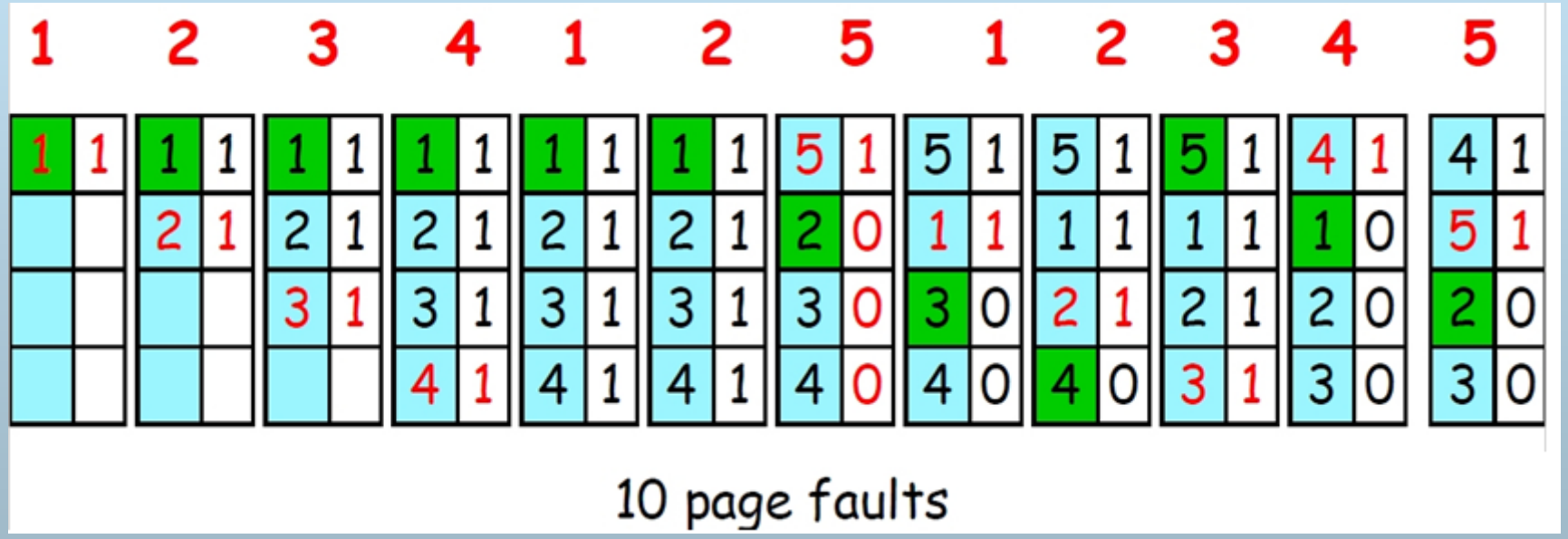
1、LRU vs. CLOCK

- LRU (Oracle, Sybase, Informix)
  - 当Pin-count为0时，frame放入替换队列
  - 选择队列头的frame替换
- Clock (MS SQL Server) 使用指针实现second chance
  - N个frame组成环形，current指针指向当前frame；每个frame有一个referenced位，初始为1；
  - 当需要置换页时，从current开始检查，若pin-count>0，current增加1；若referenced已启动（=1），则关闭它（=0）并增加current（保证最近的不被替换）；若pin-count=0并且referenced关闭（=0），则替换

LRU被scan操作污染，即对顺序访问操作性能较差，例如，buffer大小为4，对于1,2,3,4,5,1,2,3,4,5访问顺序性能差

1、LRU vs. CLOCK

- **Clock算法的一个示例**
  - **4 frames，10 requests**



2、为何不使用OS缓冲区管理？

- DBMS经常能预测访问模式(Access Pattern)
  - 可以使用更专门的缓冲区替换策略
  - 有利于pre-fetch策略的有效使用
- DBMS需要强制写回磁盘能力（如WAL），OS的缓冲写回一般通过记录写请求来实现（来自不同应用），实际的磁盘修改推迟，因此不能保证写顺序

DBMS在修改数据时需要先写日志

三、缓冲区管理器的实现

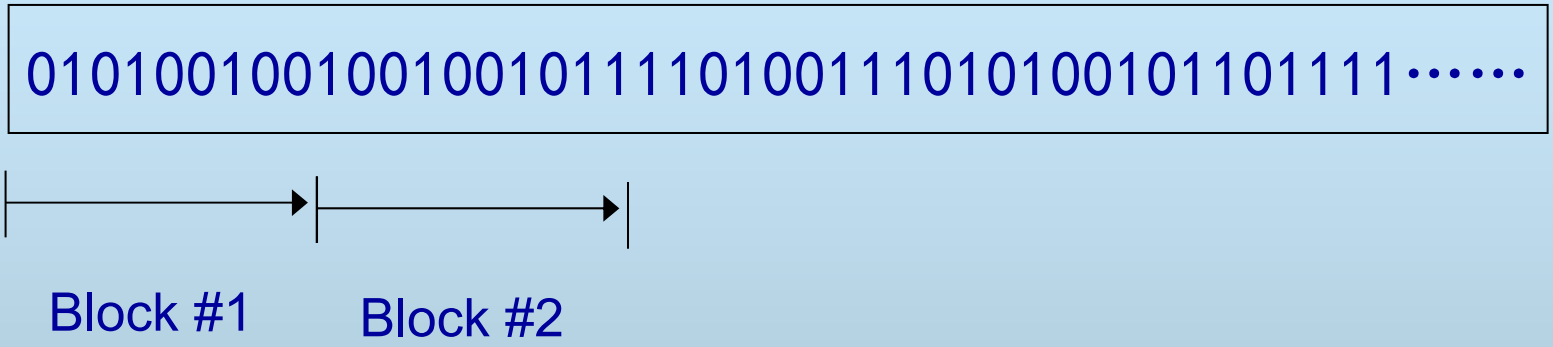

record请求
文件、记录、索引管理：
Insert_Record, Delete_Record, Create_Table, Drop_Table,
Create_Index, ……
frame请求
Buffer Manager：
FindFrame, FixPage, FixNewPage, SetDirty, SetVictim, ……
page请求
Storage Manager 将字节流抽象成磁盘块

1、错误的记录操作实现例子

- 例如，插入记录
  int insert_record(DBFILE*, DBRECORD)
  - ……
  - fopen()
  - fseek()
  - fwrite()
  - ……
- 没有DBMS自己的缓冲区管理和存储管理
- 直接基于文件系统，使用了FS的缓冲管理
  - 不能保证WAL
  - 不利于查询优化
  - 不适应应用需求

2、Block vs. Disk File

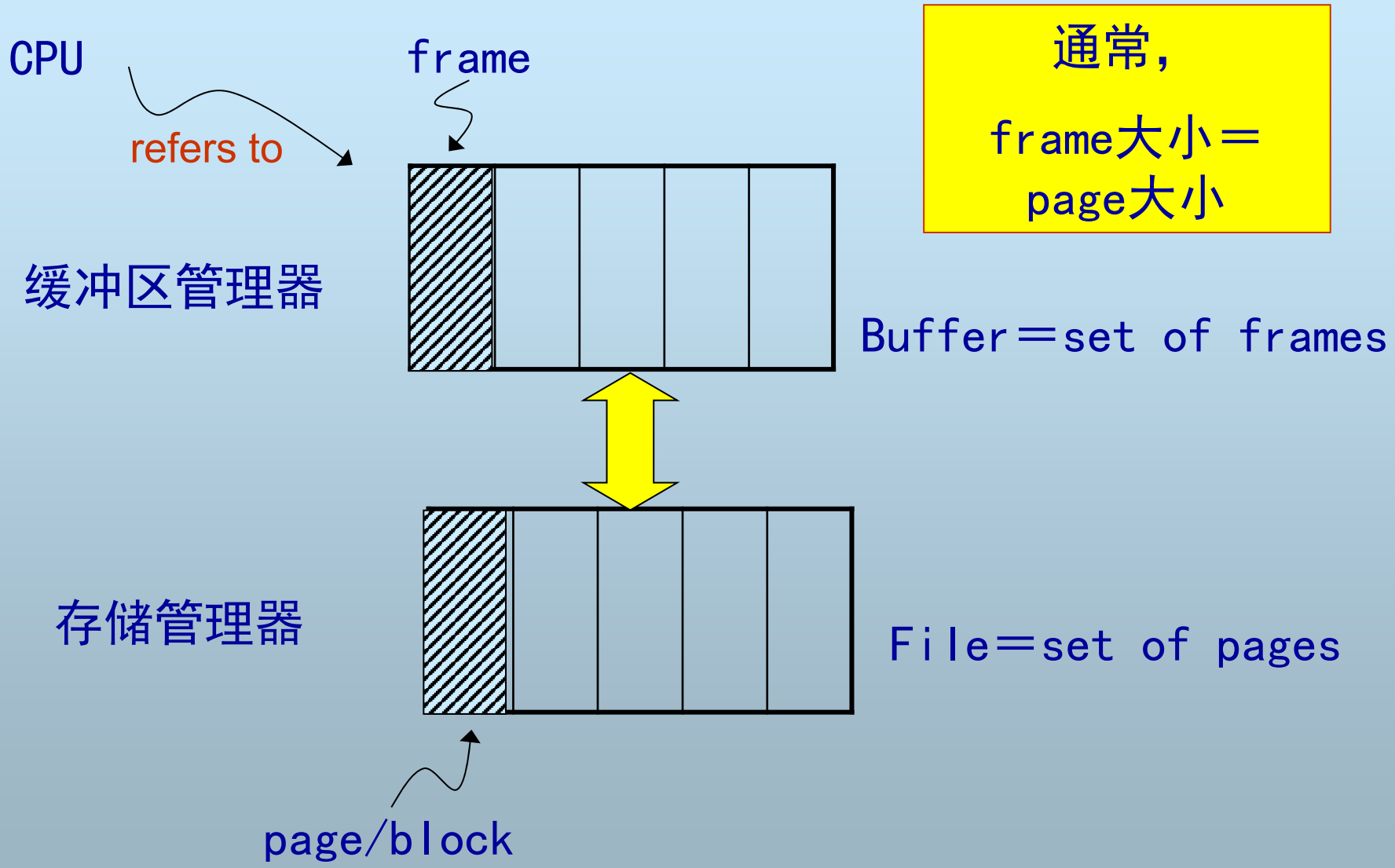
- **Disk File** 字节流文件，划分为磁盘块



访问目录块查询块的偏移量

文件存储在磁盘上的物理形式是bits/bytes，block是由OS或DBMS软件对文件所做的抽象，这一抽象是通过控制数据在文件中的起止offset来实现的

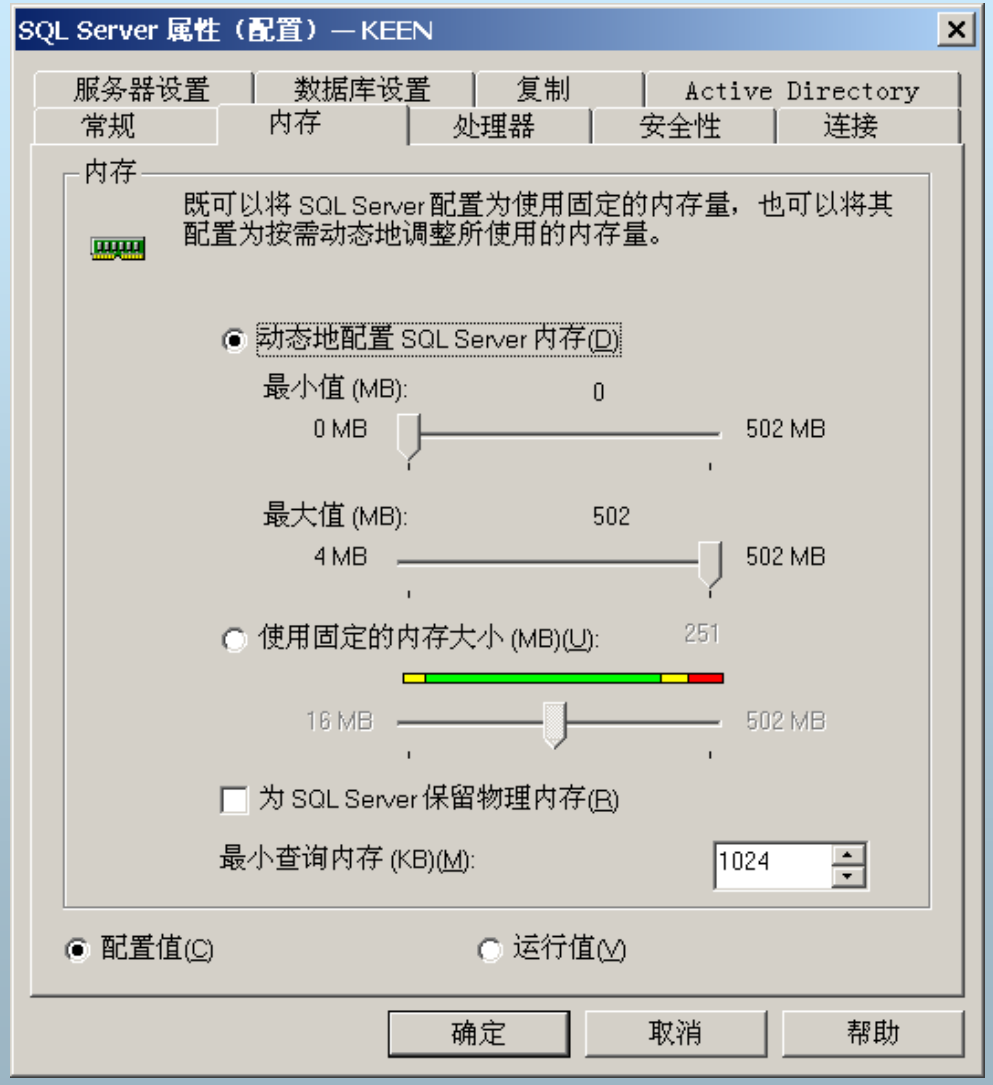
3、Buffer vs. Disk File



4、Buffer Size

- 设计DBMS时应是一个可变的输入参数
- 通常DBMS允许用户自行配置

4、Buffer Size



配置实例

5、Buffer的存储结构

- **Buffer是一个frame的列表，每个frame用于表示和存放一个磁盘块**

Buffer的存储结构定义示例

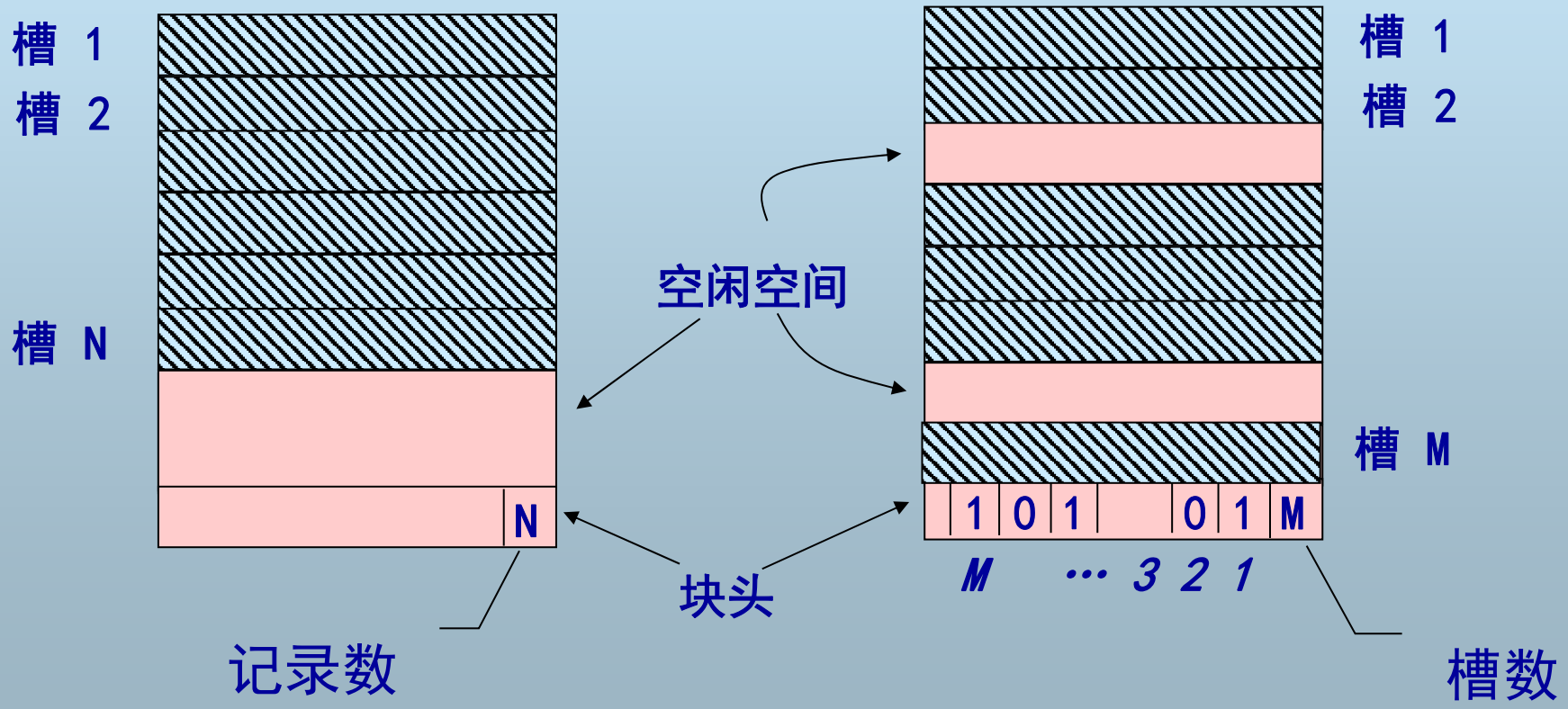
```
#define FRAMESIZE  4096
struct bFrame
{
  Char field [FRAMESIZE ];
};
```

```
#define BUFSIZE  1024 // frame数目

bFrame buf[BUFSIZE];
//也可以是用户配置的值
```

6、Page/block的一般存储格式

- 对于定长记录
  - 记录地址**rid**通常使用*<块号，槽号>*表示

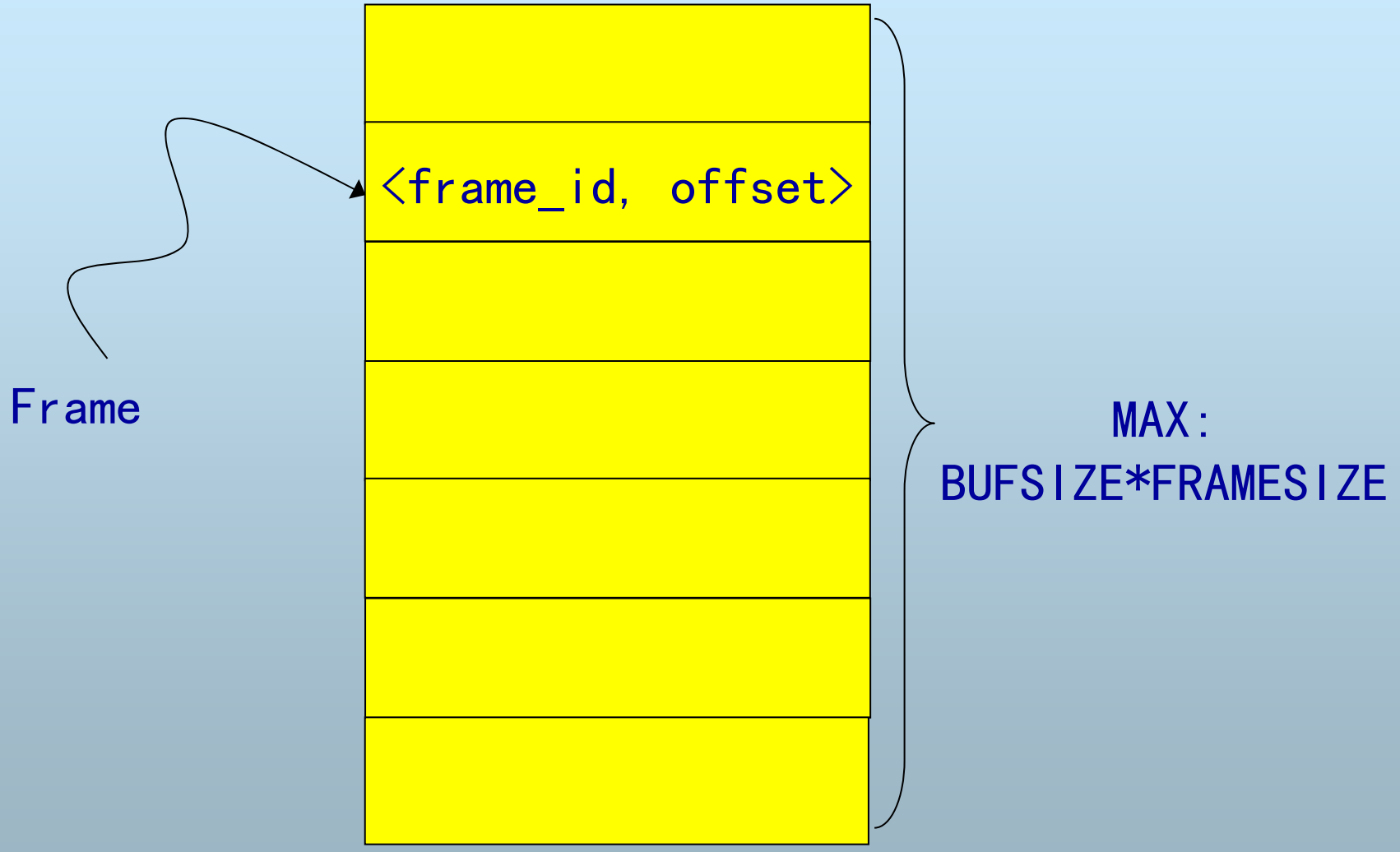


6、Page/block的一般存储格式

- **Record的存储结构**

```
struct Record {
    int page_id;
    int slot_num;
};
```

7、Buffer中的Frame存储结构



7、Buffer中的Frame存储结构

```
struct Frame{

        int frame_id;
        int offset;
} ;
```

8、Buffer中Frame的查找

- 读磁盘块时：根据page_id确定在Buffer中是否已经存在frame
- 写磁盘块时：要根据frame_id快速找到文件中对应的page_id

8、Buffer中Frame的查找

- 首先，要维护**Buffer**中所有**frame**的维护信息（**Buffer Control Blocks**），如

```
struct BCB
 {
  BCB();
  int page_id;
  int frame_id;
  int count;
  int time;
  int dirty;
  BCB * next;
};
```

8、Buffer中Frame的查找

- 建立frame-page之间的索引
- 若用Hash Table，需要建立2个
  - BCB hTable[BufferSize] //page 2 frame
  - int hTable[BufferSize] //frame 2 page

类似于HashMap的数据结构

实验需实现

一个简单的Hash Function例子

H(k)=(page_id)%(buffersize)

9、Buffer Manager的基本功能

- **FixPage(int page_id)**
  - 将对应page_id的page读入到buffer中。如果buffer已满，则需要选择换出的frame。
- **FixNewPage()** 实验时，先构建堆文件，update时先读块再写块
  - 在插入数据时申请一个新page并将其放在buffer中
- **SelectVictim()**
  - 选择换出的frame_id
- **FindFrame(int page_id)**
  - 查找给定的page是否已经在某个frame中了
- **SetDirty(int frame_id)**

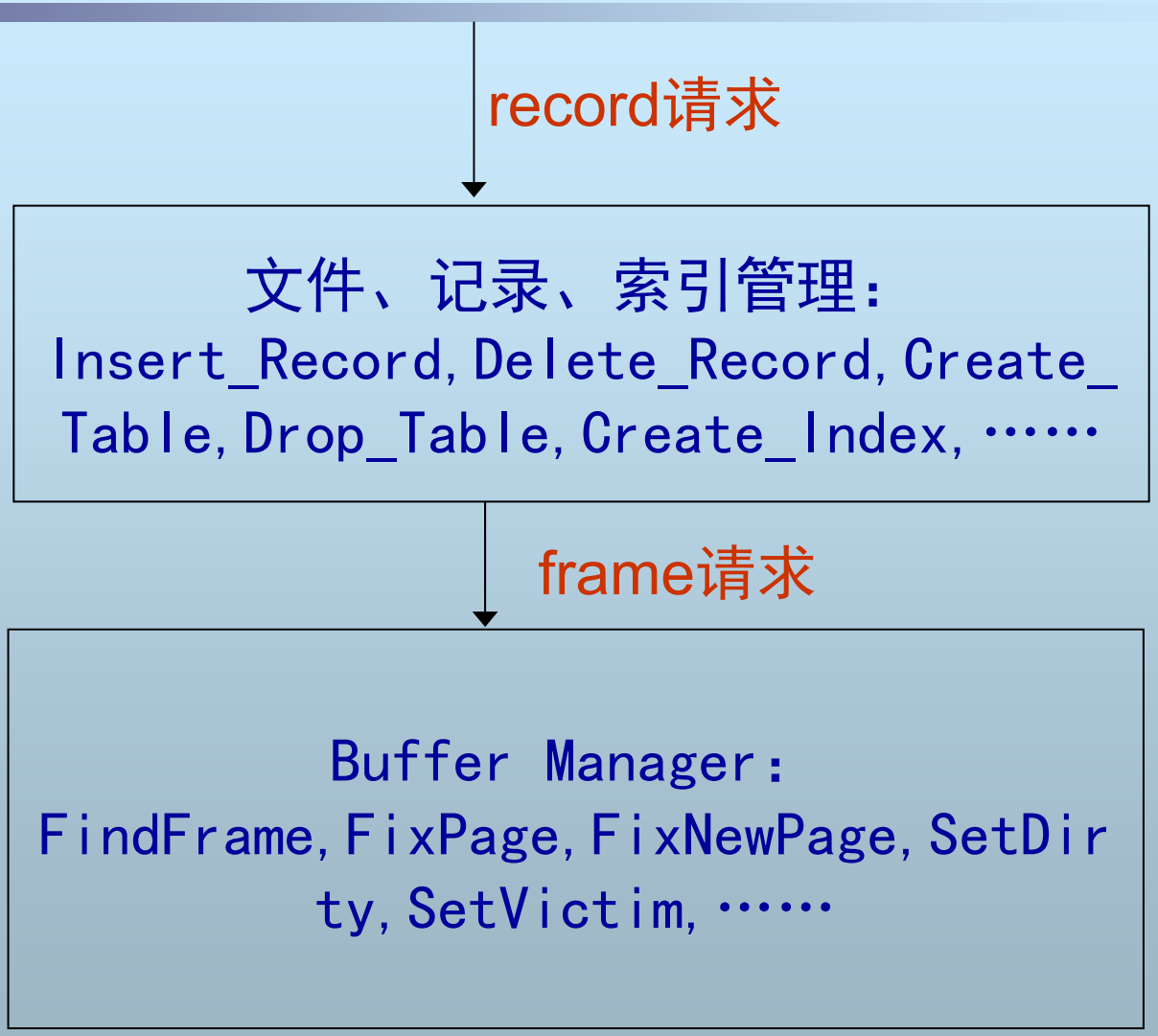
10、数据库文件的一般组成

- 数据文件
  - 首块在Insert_Record时创建（调用Buffer Manager的FixNewPage），一般块号从0开始
- 系统目录文件
  - 首块一般Create_Table时创建（调用Buffer Manager的FixNewPage）

Note:

所有数据和元数据操作都唯一通过Buffer Manager来请求page

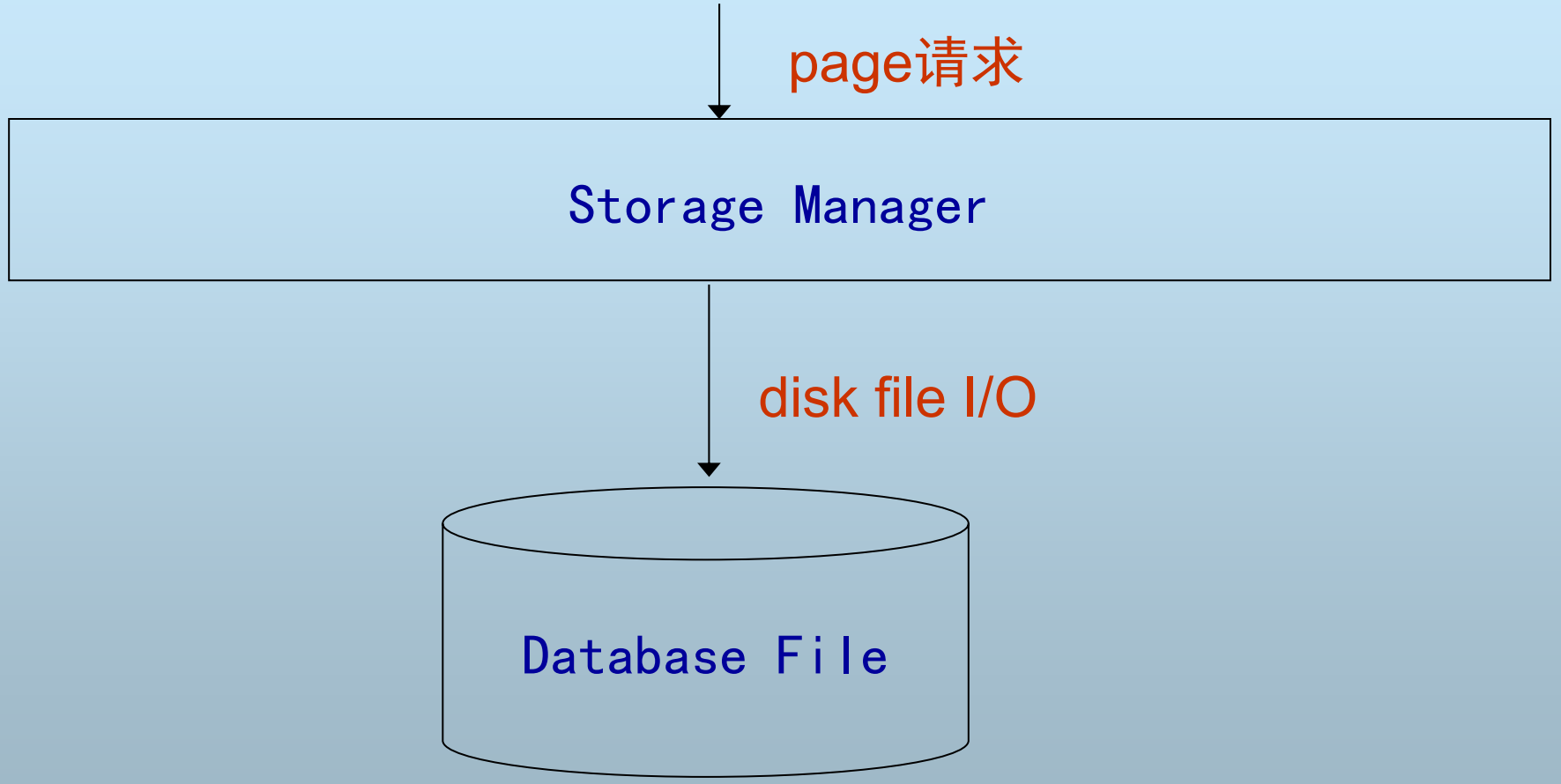
11、文件记录操作与Buffer Manager



12、存储管理器

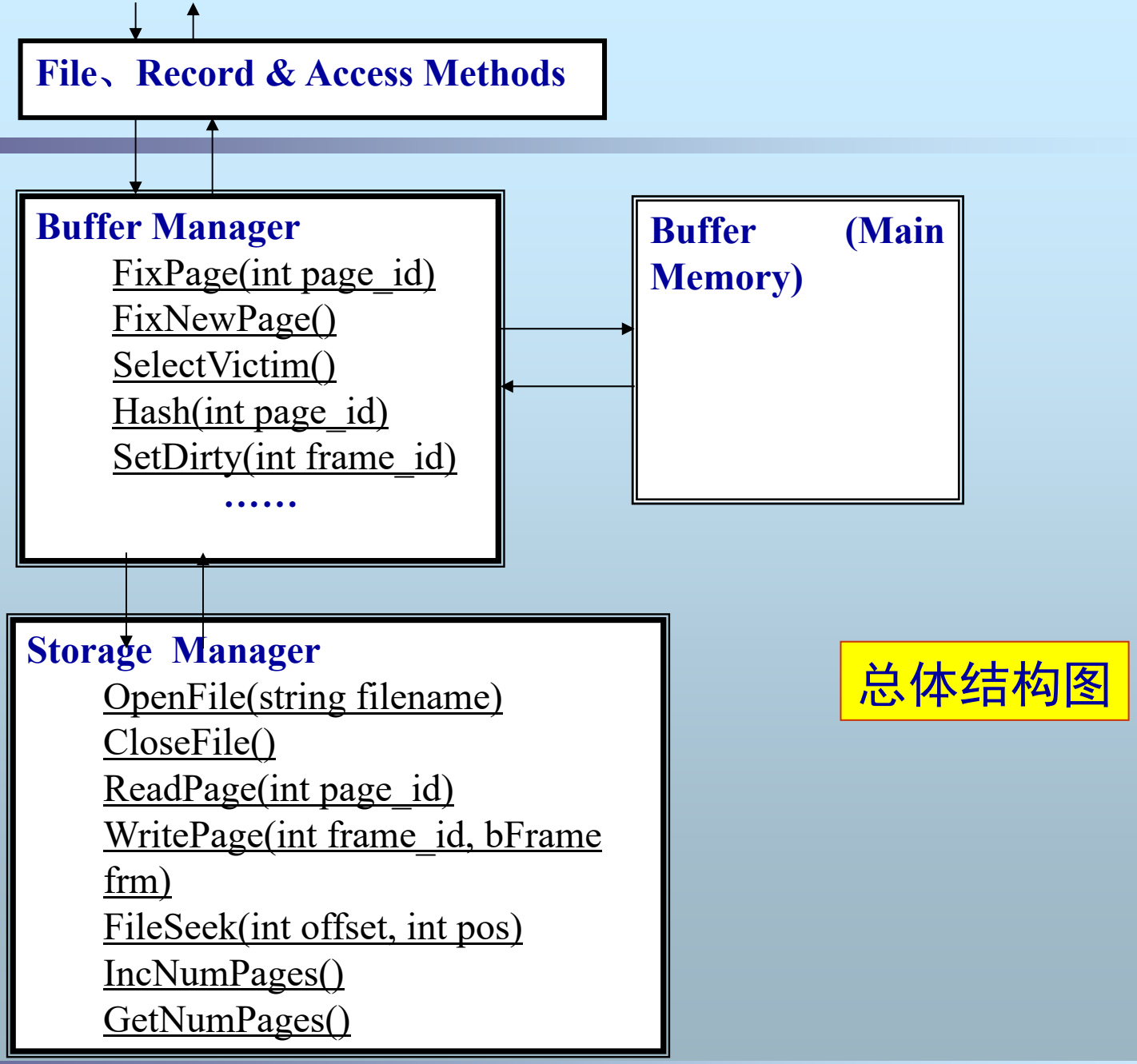

record请求
文件、记录、索引管理：
Insert_Record, Delete_Record, Create_Table, Drop_Table, Create_Index, ……
frame请求
Buffer Manager：
FindFrame, FixPage, FixNewPage, SetDirty, SetVictim, ……
page请求
Storage Manager

12、存储管理器



13、存储管理器功能

- 从磁盘中存取物理数据，为Buffer Manager提供Page抽象
  - OpenFile/CloseFile
  - ReadPage/WritePage
  - FileSeek
  - GetNumPages
  - IncreaseNumPages
  - ……


File、Record & Access Methods
Buffer Manager
FixPage(int page_id)
FixNewPage()
SelectVictim()
Hash(int page_id)
SetDirty(int frame_id)
……
Buffer (Main Memory)
Storage Manager
OpenFile(string filename)
CloseFile()
ReadPage(int page_id)
WritePage(int frame_id, bFrame frm)
FileSeek(int offset, int pos)
IncNumPages()
GetNumPages()

总体结构图

总结

- 缓冲区结构
- 缓冲区置换算法
- 缓冲区管理的实现