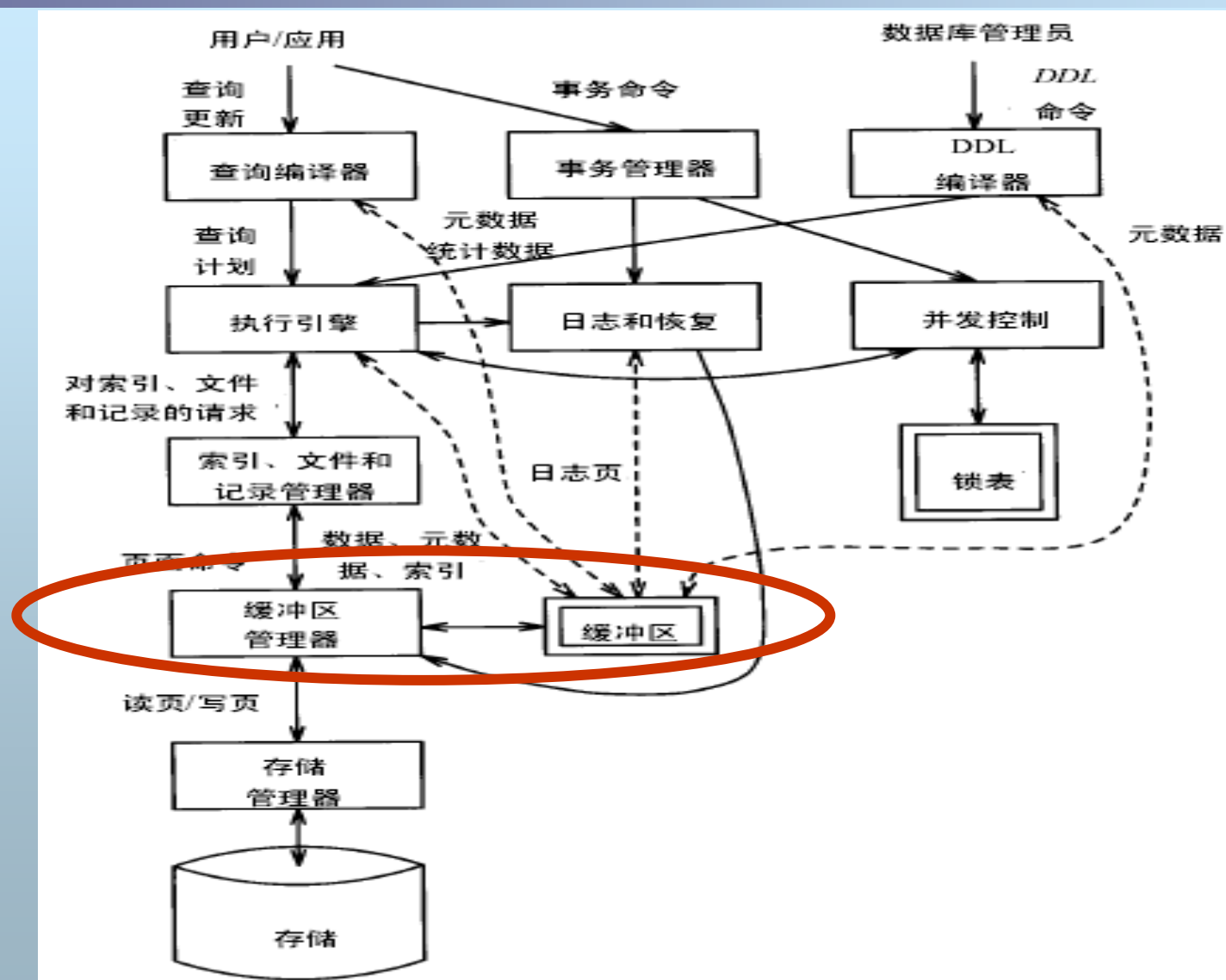


# Buffer Management



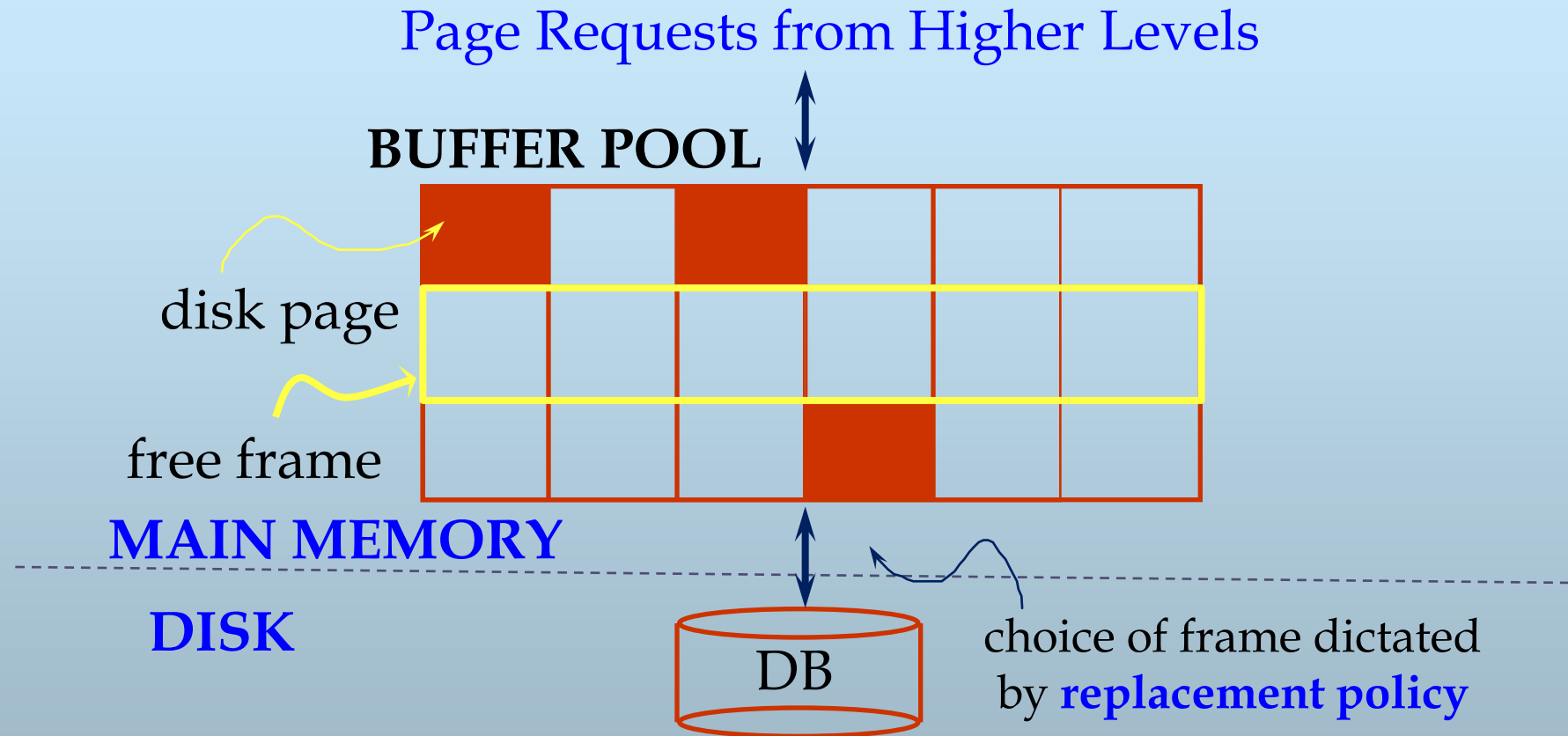
**Idea:** Minimize the count of disk I/Os by keeping likely-to-be-requested pages in memory (buffer frames)



# 主要内容

- 缓冲区结构
- 缓冲区置换算法
- 缓冲区管理的实现

# 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

## 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.

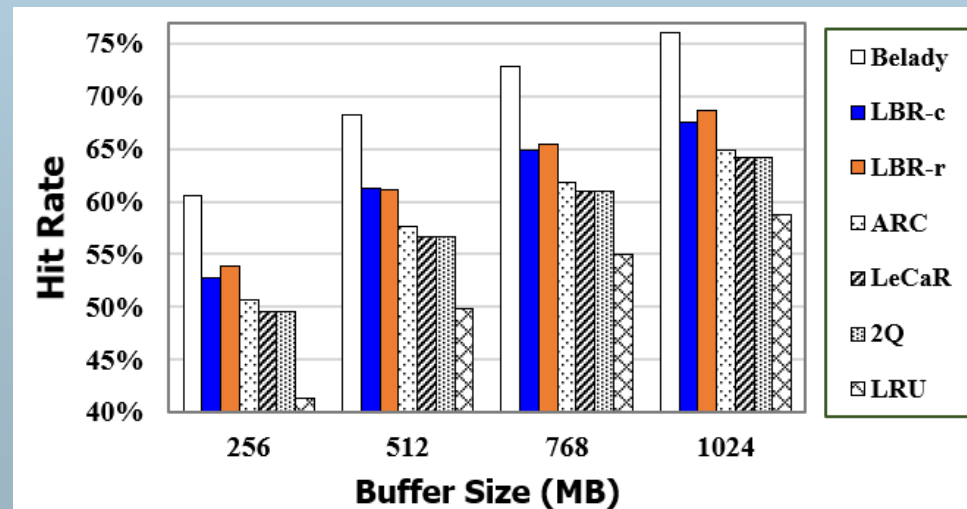
## 二、缓冲区替换策略

- 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 pattern*.



## 二、缓冲区替换策略

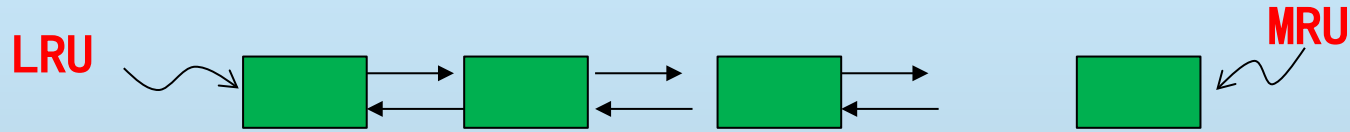
- 理论最优算法：**OPT算法**
  - 也称为**Belady's算法**
  - 理论上最佳的页面置换算法。它每次都置换以后永远也用不到的页面，如果没有则淘汰最久以后再用到的页面。
  - **OPT算法必须预先知道全部的页面访问序列，而这在实际DBMS/OS中是无法实现的，因此仅有理论意义。**
- 但**OPT算法可以在实验中作为算法性能上界加以对比**



# 1、LRU

## ■ LRU (Oracle, Sybase, Informix)

- 所有frame按照最近一次访问时间排列成一个链表



- 基于时间局部性(Temporal Locality) 假设：越是最近访问的在未来被访问的概率越高. 总是替换LRU端的frame

## ■ Pros

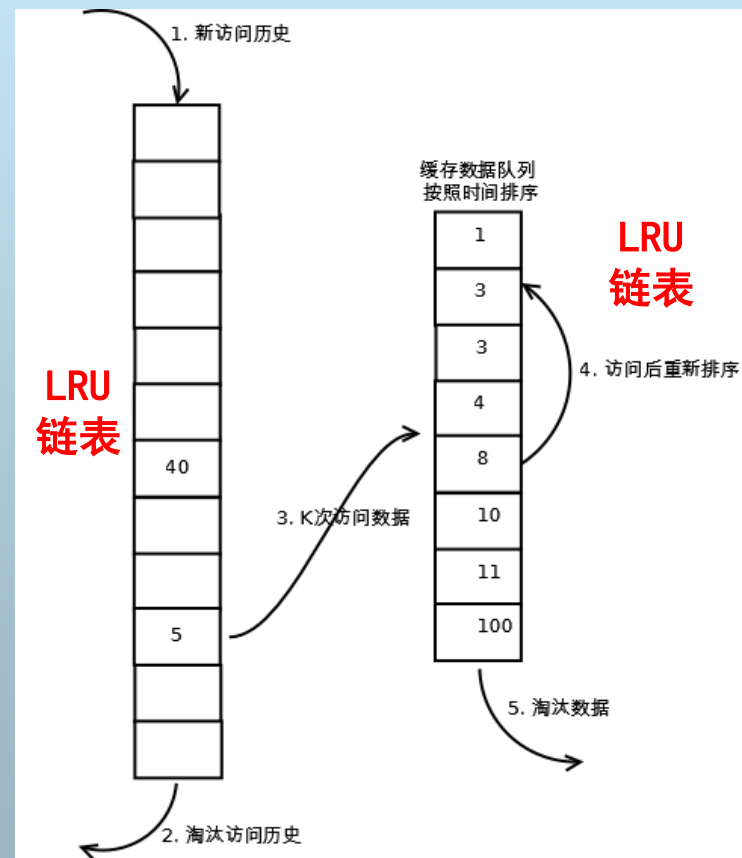
- Works well for repeated access to popular pages
- $O(1)$  complexity for victim selection

## ■ Cons:

- 缓存污染(Sequential flooding): LRU + repeated sequential scans.
- 每次访问都需要修改LRU链表, 并发性能差
- 如果访问不满足时间局部性, 则性能较差
- 只考虑最近一次访问, 不考虑访问频率

## 2、LRU-K

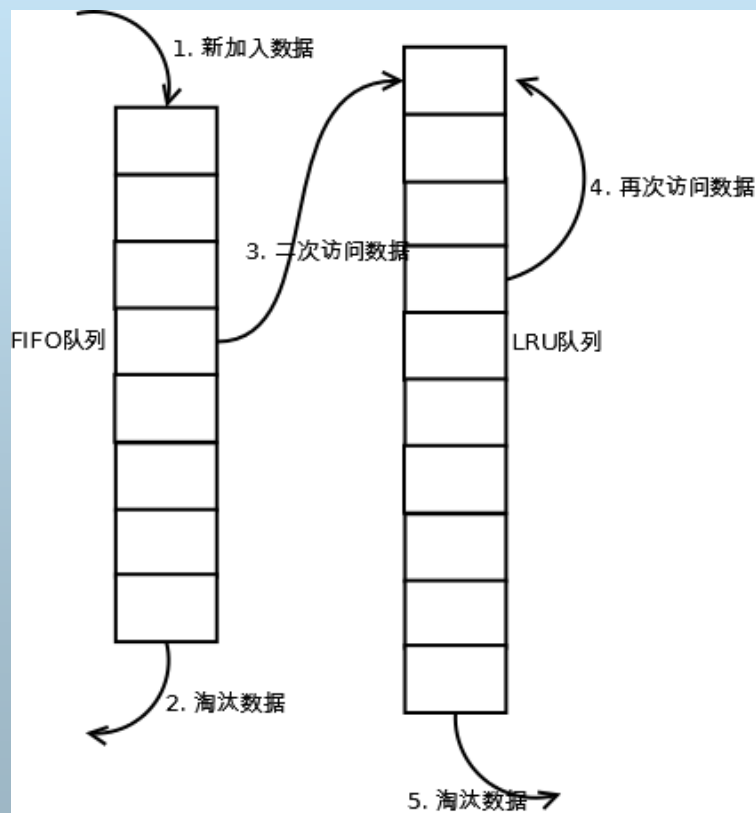
- LRU不考虑frame的访问频率，不合理
- LRU-K: 如果某个frame的访问次数达到了K次以上，则应当尽量不置换
  - 维护2个LRU链表
    - ◆ 1个是访问次数小于K次的
    - ◆ 1个是访问次数K次以上的
  - 优先按照LRU策略置换小于K次的链表
  - 保证高频访问的页能够尽量在buffer中
  - 实验表明
    - ◆ K并非越大越好，LRU-2 性能较好
  - 缺点：需要额外记录访问次数



E. O'Neil, P. O'Neil, G. Weikum: The LRU-K Page Replacement Algorithm for Database Disk Buffering, SIGMOD 1993, pp. 297-306

# 3、2Q

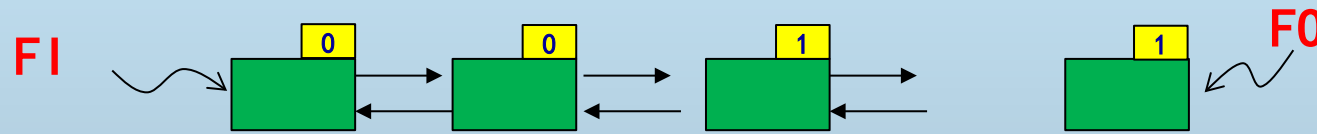
- 与LRU-2类似，不同之处在于访问1次的队列采用FIFO，而不是LRU



T. Johnson, D. E. Shasha: 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. VLDB 1994: 439-450

# 4、Second-Chance FIFO

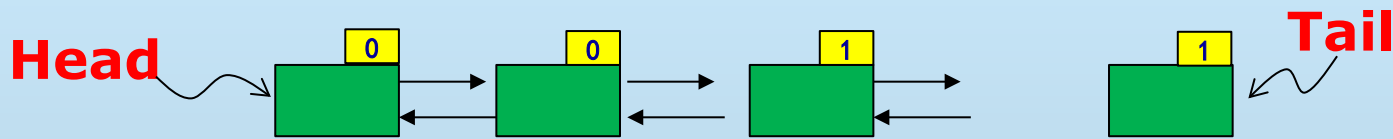
- 所有frame组成FIFO链表，每个frame附加一个bit位，初始为0。当FO页第一次被选中置换时置为1，并移到FI端。只有bit位为1的FO端的页才被选中置换。



- 相当于每个frame给了两次置换机会，避免高频访问但最近一轮没有被访问的frame被置换出buffer
- 每个frame只需要1个额外bit，空间代价很低
- 缺点：置换时需要移动多个元素，理论性能比LRU差

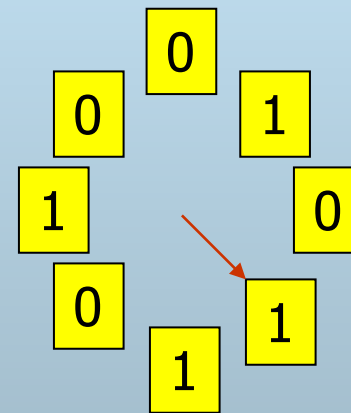
# 5、CLOCK

- 如何避免每次访问时的链表调整代价？
- Second-Chance FIFO



- Clock (MS SQL Server, PostgreSQL, GaussDB)

- 把Second-Chance FIFO组织成环形
- 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)，则替换该frame，同时current加1
- 注意：Current指针只在置换时更新，访问命中时不改变Current指针



# 6、SSD上的置换算法

- 闪存：读快写慢，写次数有限
  - 减少缓存置换中对闪存的写是一个重要指标
- SSD-aware缓存算法
  - **CFLRU** (CASES'06, CASES'21 Test of Time Award)
    - ◆ Clean-first
  - **LRU-WSR** (IEEE Trans CE'08)
    - ◆ Clean-first + cold flag
    - ◆ 置换: clean > cold dirty > hot dirty
  - **AD-LRU** (DKE'10)
    - ◆ cold LRU list + hot LRU list
    - ◆ Dynamically adjust two LRUs
  - .....

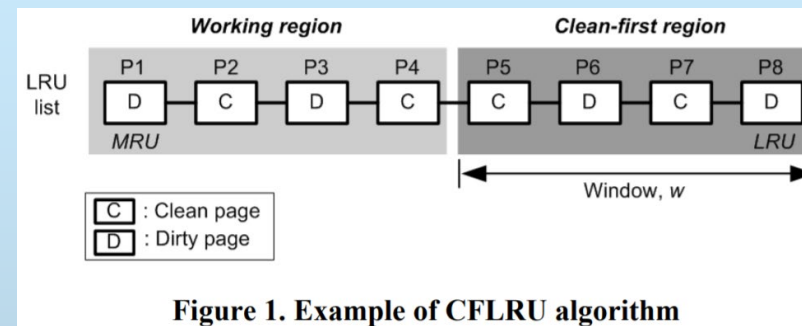
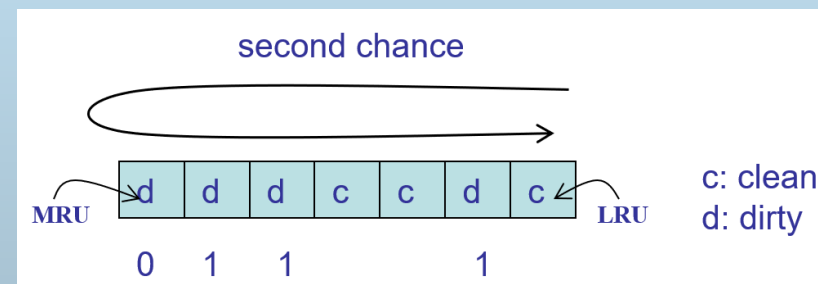


Figure 1. Example of CFLRU algorithm



	LRU	CF-LRU	LRU-WSR	AD-LRU
Recency	√	√	√	√
Cleanness*	×	√	√	√
Frequency	×	×	×	√

# 7、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**



# 三、缓冲区管理器的实现

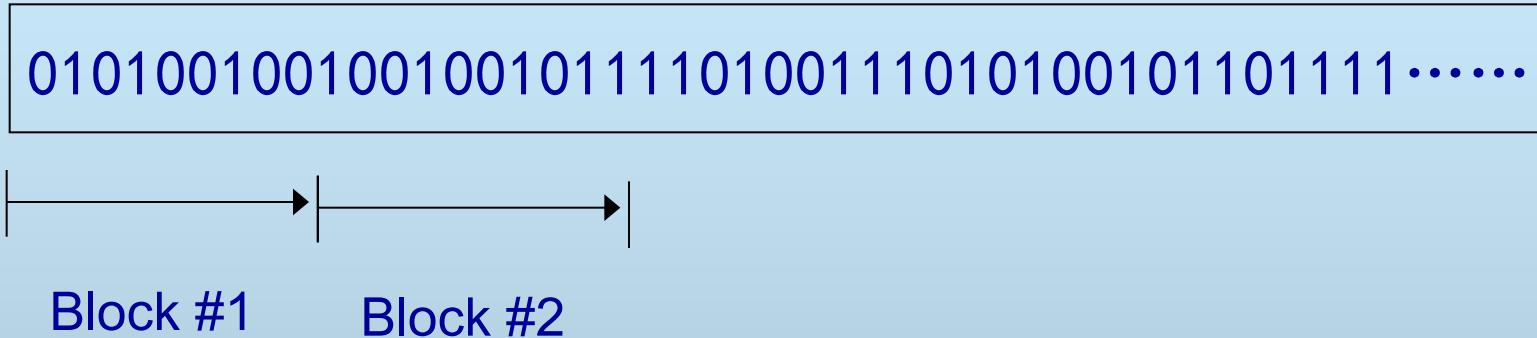


# 1、错误的记录操作实现例子

- 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

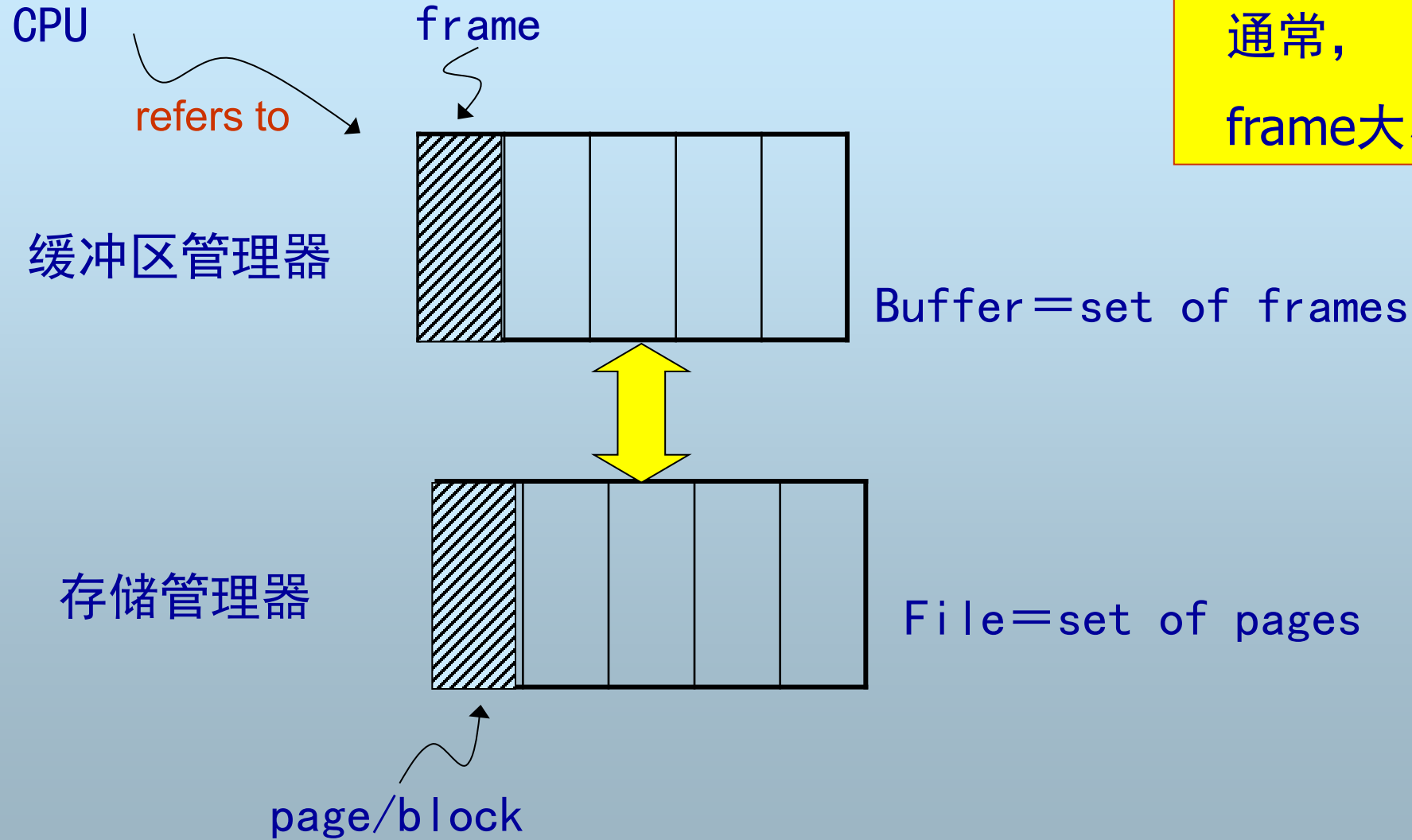
## 2、Block vs. Disk File

### ■ Disk File



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



通常,  
frame大小 = page大小

# 4、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];
//也可以为用户配置的值
```

## 5、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

## 5、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 latch; */
    int dirty;
    BCB * next;
};
```

# 5、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`

一个简单的Hash Function例子

$$H(k)=(page\_id)\%(buffersize)$$



# 6、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()

- 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)

# 7、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

# 8、文件记录操作与Buffer Manager

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, ……

# 9、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, ……

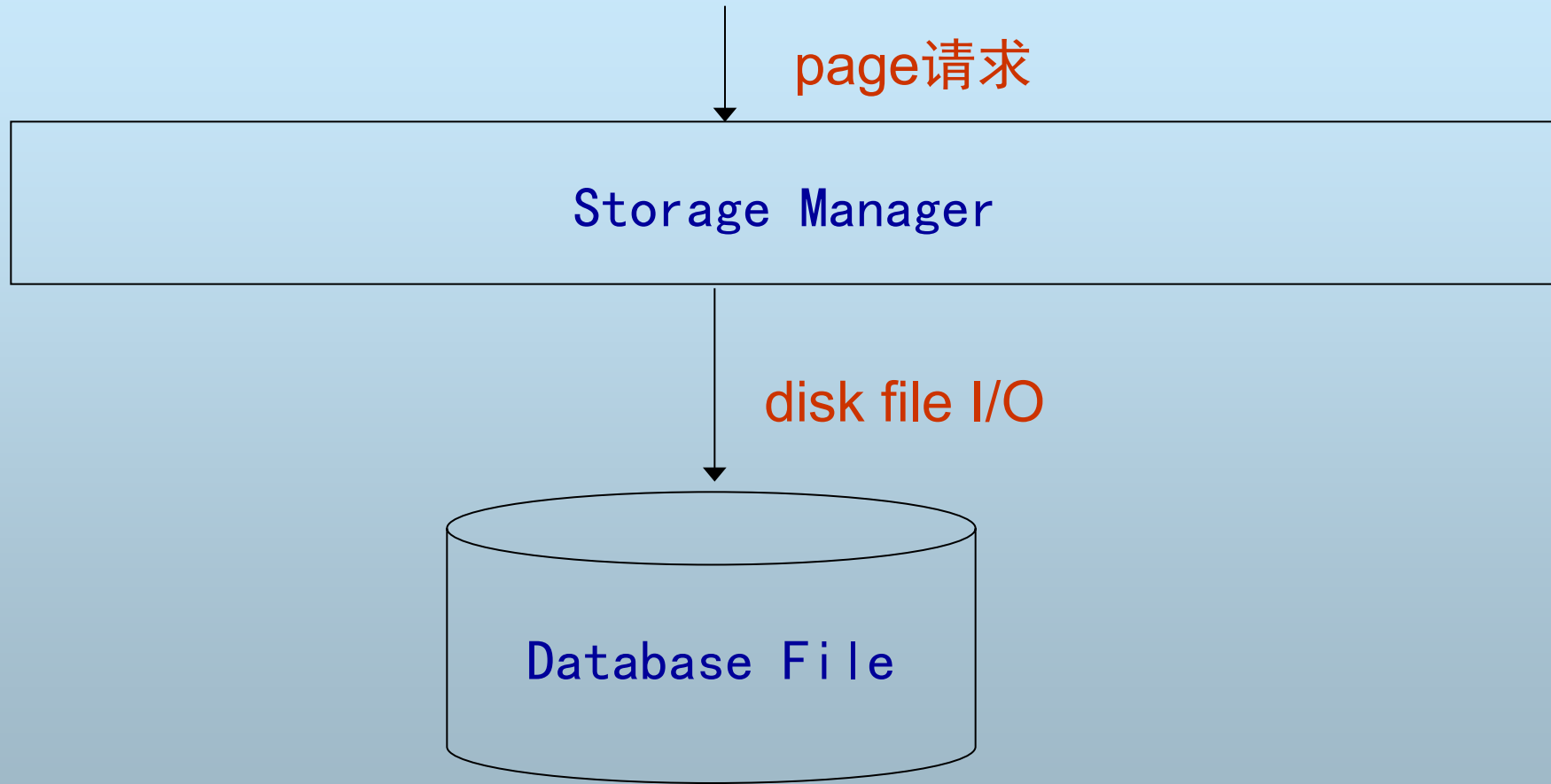
frame请求

Buffer Manager：  
FindFrame, FixPage, FixNewPage, SetDirty, SetVictim, ……

page请求

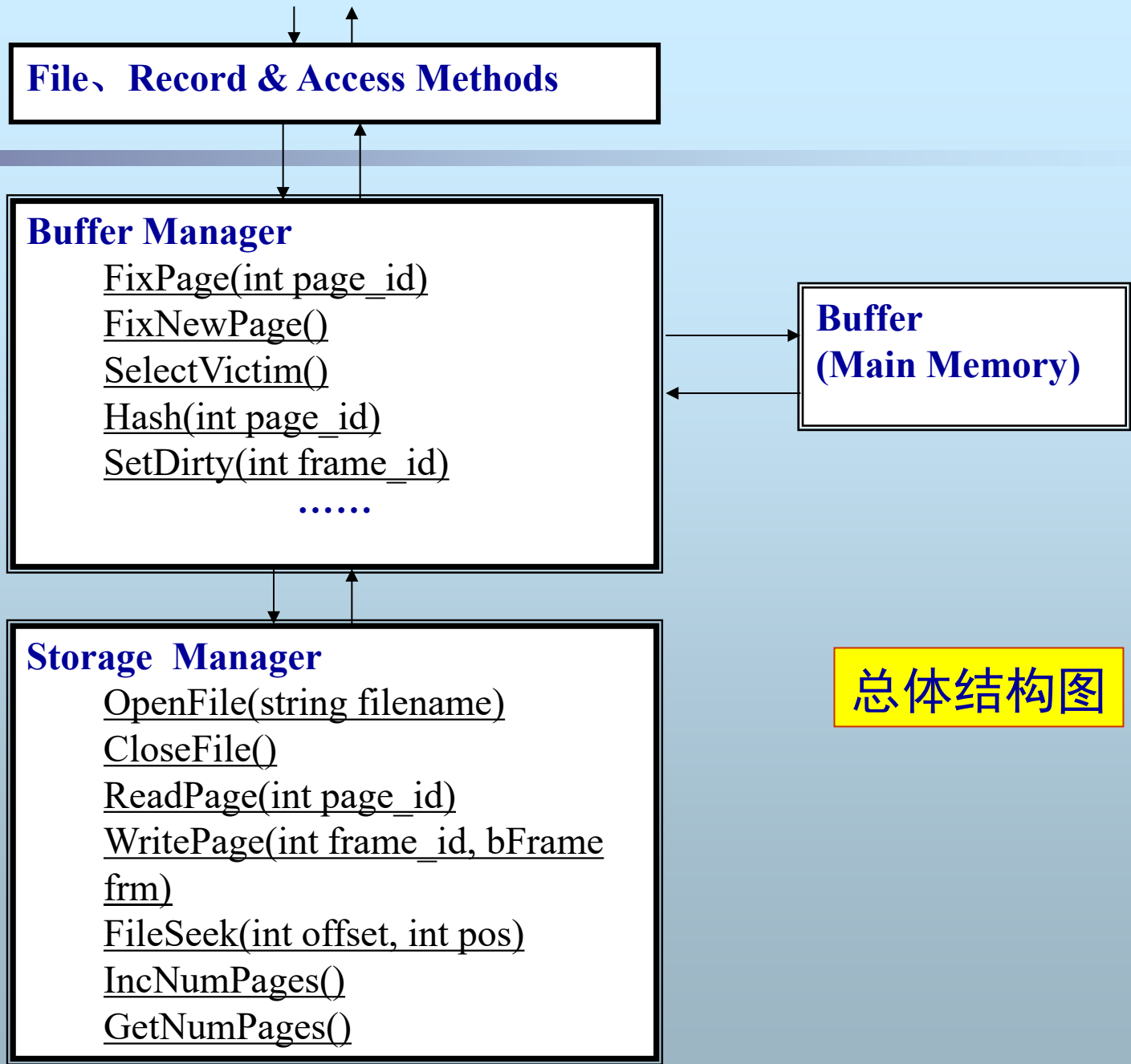
Storage Manager

# 9、存储管理器



# 10、存储管理器功能

- 从磁盘读写物理数据，为Buffer Manager提供Page抽象
  - **OpenFile/CloseFile**
  - **ReadPage/WritePage**
  - **FileSeek**
  - **GetNumPages**
  - **IncreaseNumPages**
  - .....



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