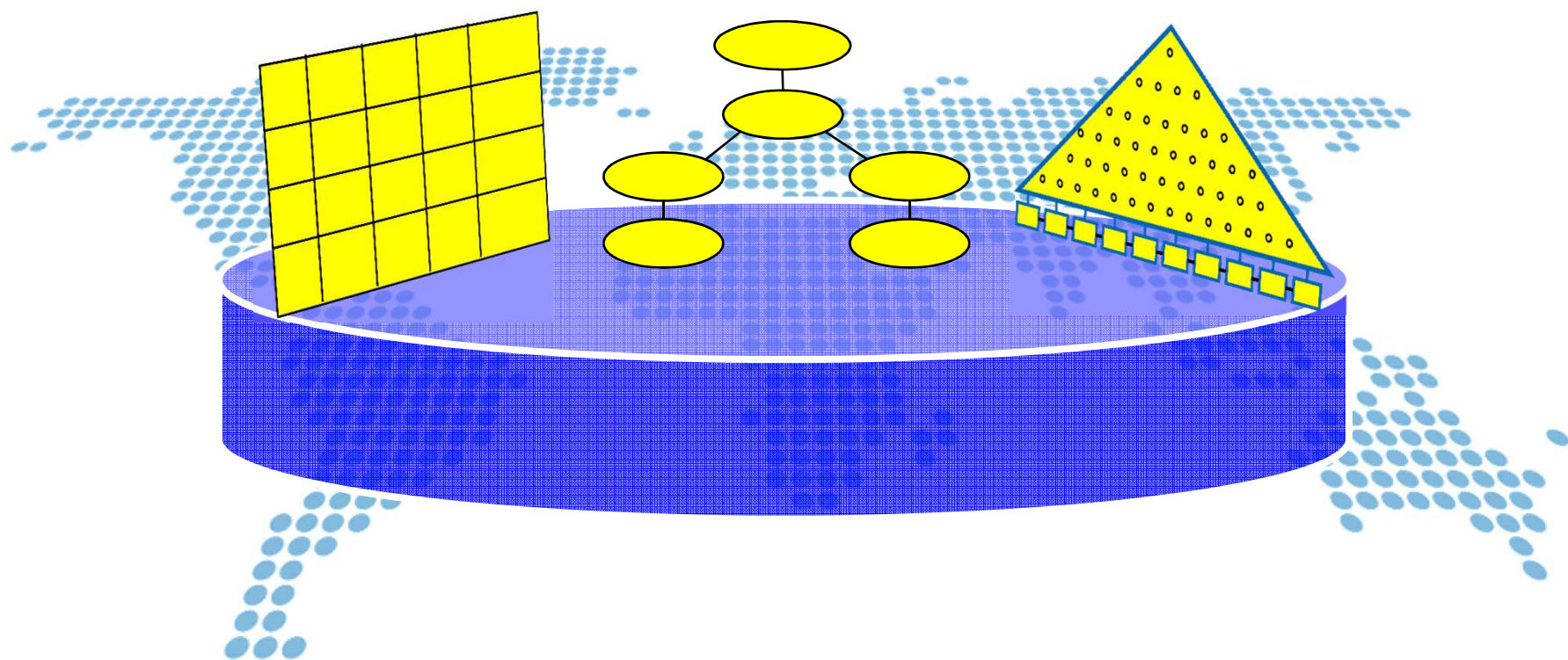


数据库系统

# 数据存储与访问路径(1)

陈世敏

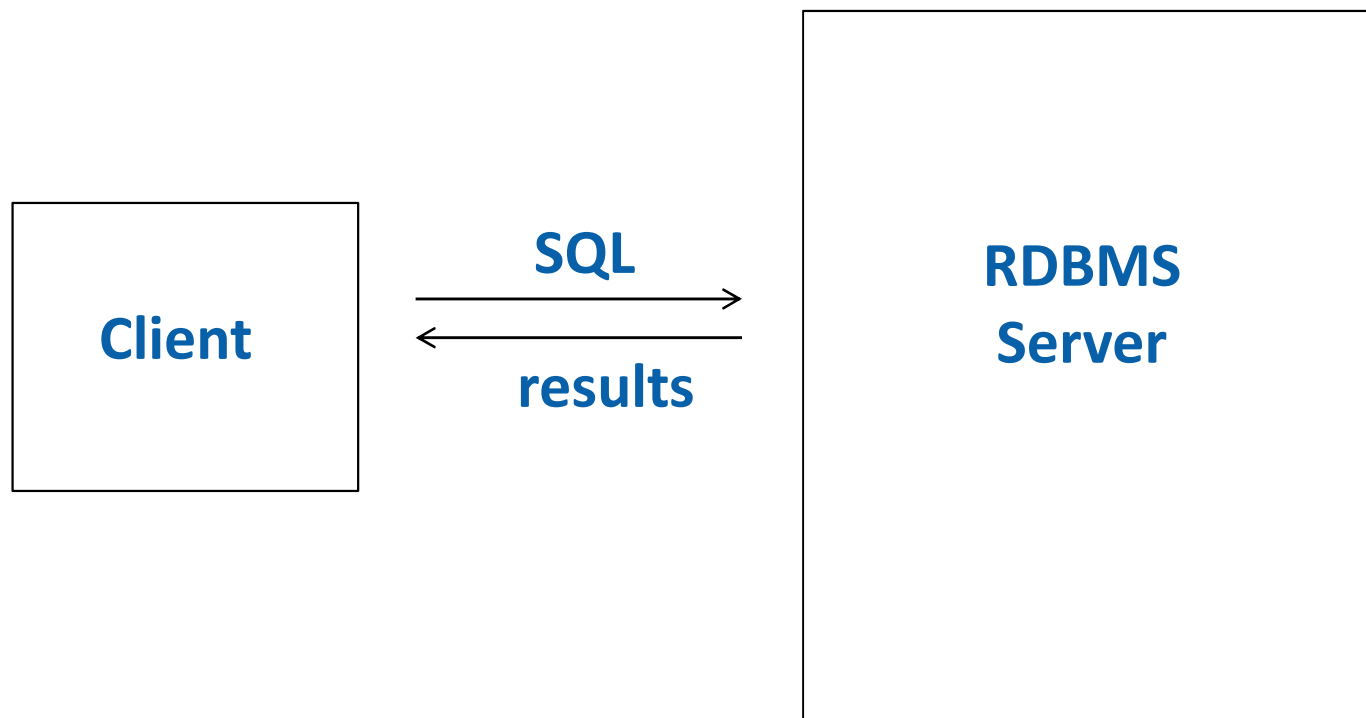
(中科院计算所)



# Outline

- 数据库系统内部架构概述
- 数据存储与访问路径概述
- 磁盘空间管理
- 记录文件格式
- 缓冲区管理

# 通常的系统为典型的Client / Server



# RDBMS的系统架构(单机)

前端

SQL Parser

Query Optimizer

后端

Execution Engine

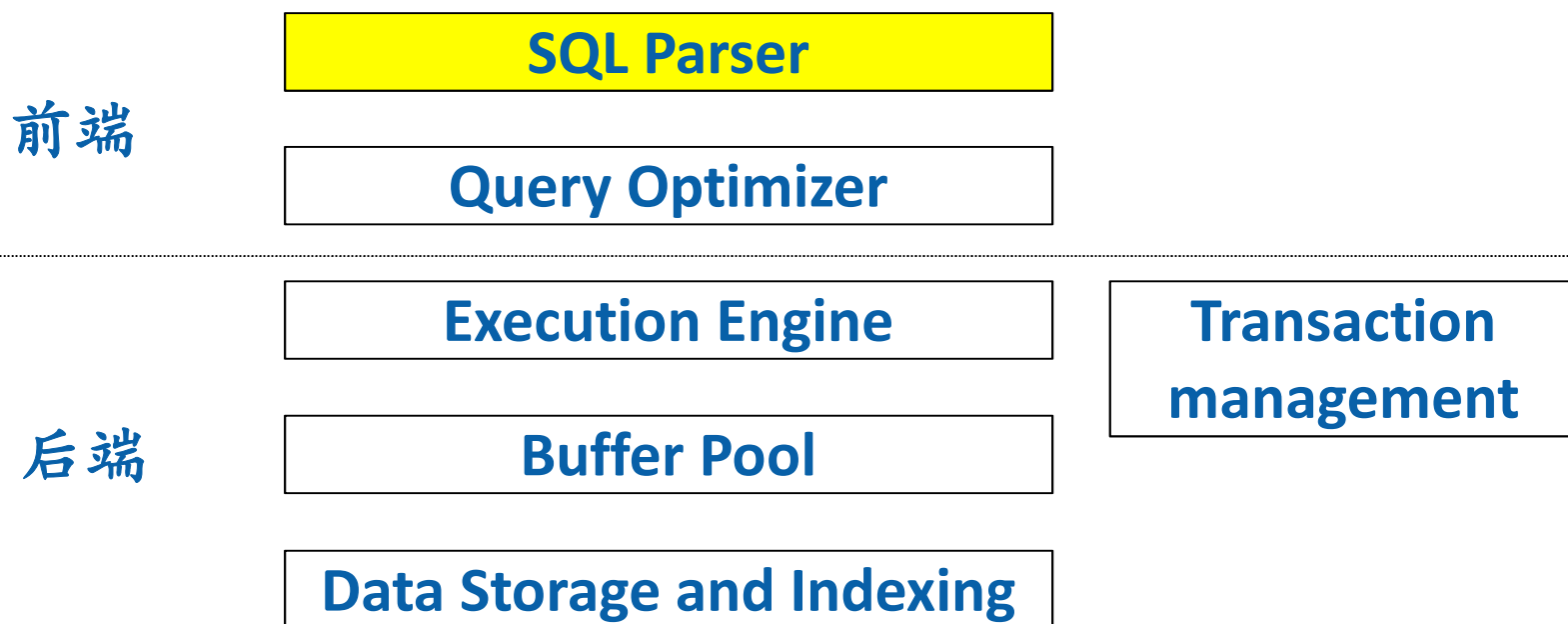
Buffer Pool

Data Storage and Indexing

Transaction  
management

# RDBMS的系统架构

- SQL 语句的程序 → 解析好的内部表达  
(例如: parsing tree)
  - 语法解析, 语法检查, 表名、列名、类型检查



# RDBMS的系统架构

- SQL内部表达 → Query Plan (执行方案)
  - 产生可行的query plan
  - 估计query plan的运行时间和空间代价
  - 在多个可行的query plans中选择最佳的query plan

前端

SQL Parser

Query Optimizer

后端

Execution Engine

Transaction  
management

Buffer Pool

Data Storage and Indexing

# RDBMS的系统架构

- Data storage and indexing

- 如何在硬盘上存储数据
- 如何高效地访问硬盘上的数据
- ☞ 最后一节课会介绍内存数据库

前端

SQL Parser

Query Optimizer

后端

Execution Engine

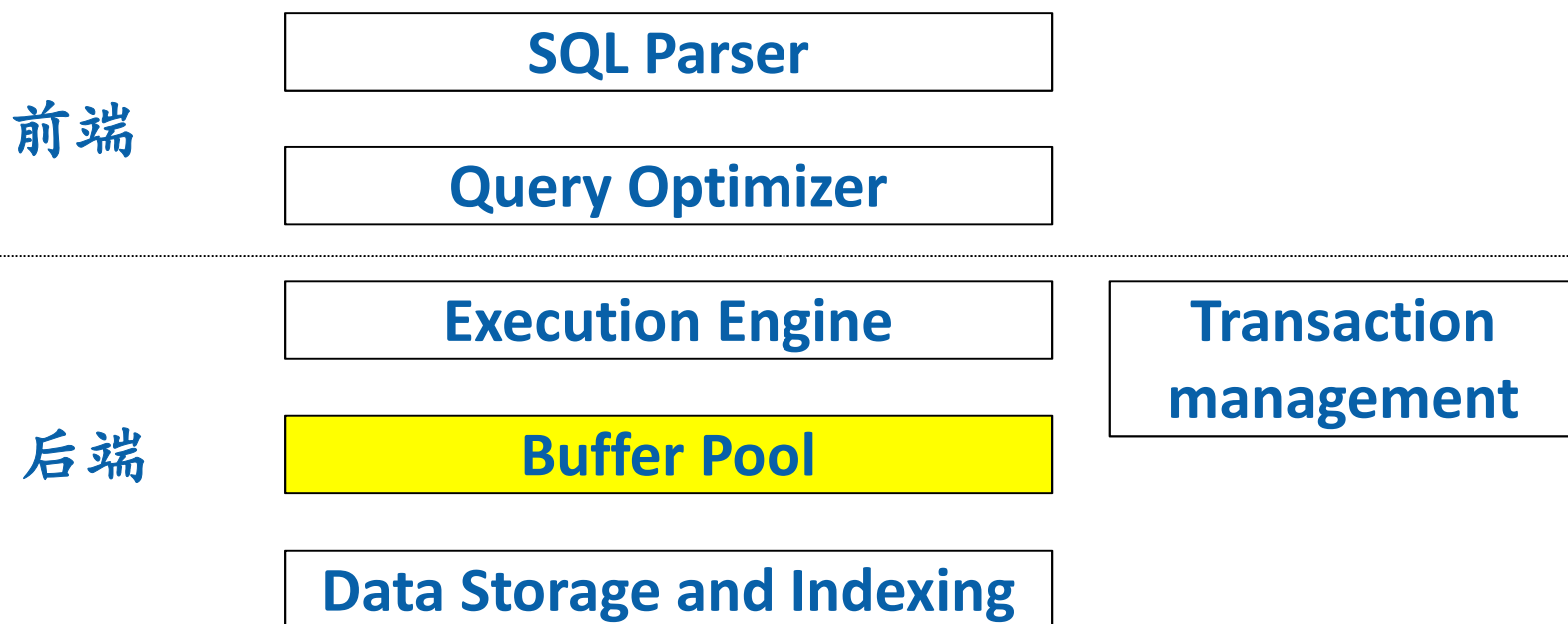
Buffer Pool

Transaction  
management

Data Storage and Indexing

# RDBMS的系统架构

- Buffer pool: 在内存中缓存硬盘的数据
  - 数据重复使用
  - 优化写操作





# RDBMS的系统架构

- query plan → SQL语句的结果

- 根据query plan, 完成相应的运算和操作
- 数据访问
- 关系型运算的实现

前端

SQL Parser

Query Optimizer

后端

Execution Engine

Buffer Pool

Data Storage and Indexing

Transaction  
management

# RDBMS的系统架构

- Transaction management: 事务管理

- 目标是实现ACID

- Atomicity, Consistency, Isolation, Durability

- 进行logging写日志, locking加锁

- 保证并行transactions事务的正确性

前端

SQL Parser

Query Optimizer

后端

Execution Engine

Buffer Pool

Data Storage and Indexing

Transaction  
management

# ACID: DBMS保证事务的ACID性质

- **Atomicity**

- all or nothing
- 要么完全执行，要么完全没有执行

- **Consistency**

- 从一个正确状态转换到另一个正确状态  
(正确指: constraints, triggers等)

- **Isolation**

- 每个事务与其它并发事务互不影响

- **Durability**

- Transaction commit后，结果持久有效，  
crash也不消失



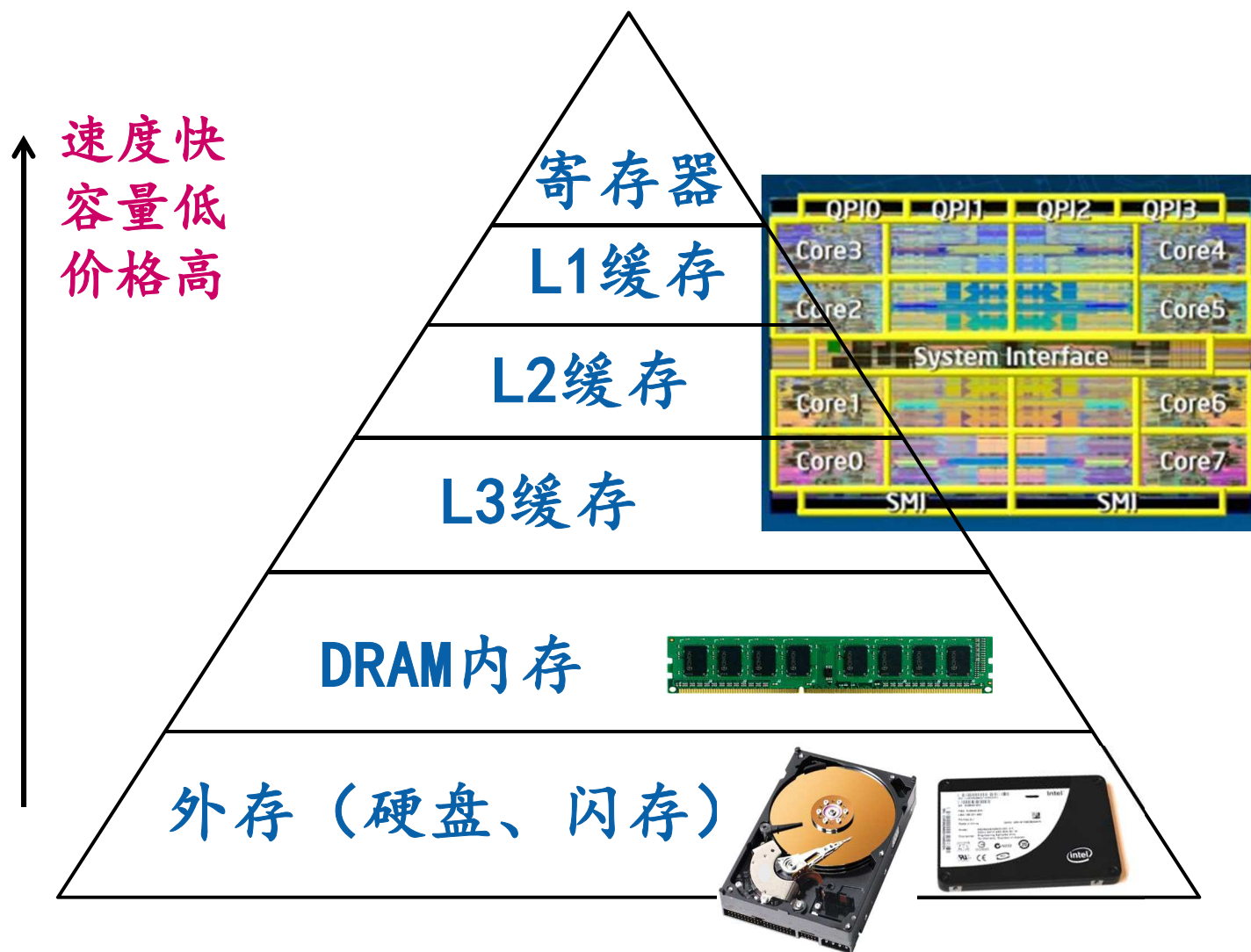
**Concurrency  
Control**

**Crash  
Recovery**

# Outline

- 数据库系统内部架构概述
- 数据存储与访问路径概述
  - 存储层次
    - 存储介质：磁盘、固态硬盘等
    - 磁盘阵列
    - 操作系统支持
    - 数据与索引
- 磁盘空间管理
- 记录文件格式
- 缓冲区管理

# 存储层次结构



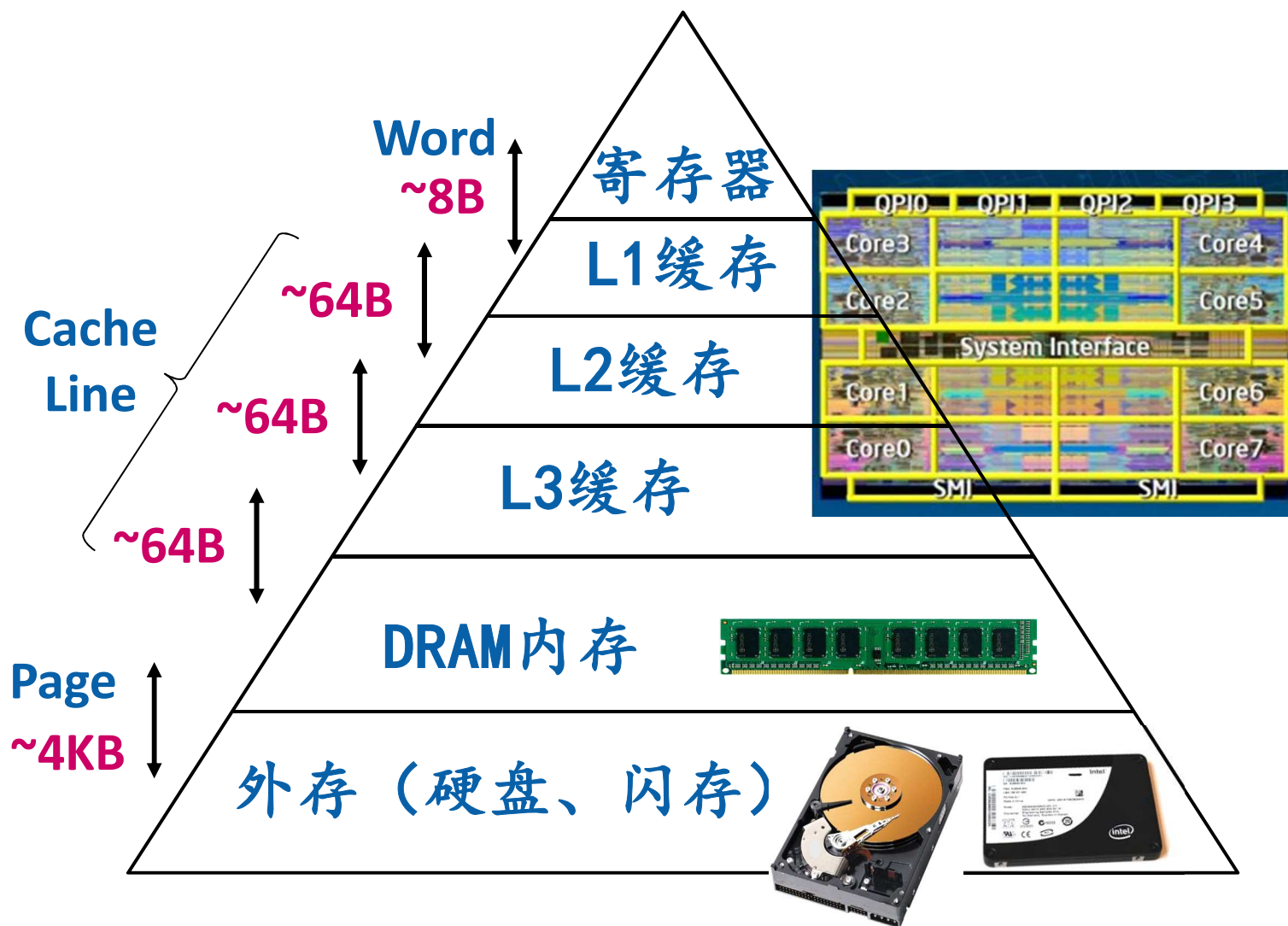
# 存储层次比较(中档服务器)

	容量	单位价格	读写延时	读写带宽
寄存器 (Register)	100B~1KB	\$100/KB	<1ns	-
高速缓存 (Cache, SRAM)	~10MB	\$10/MB	1~10ns	~TB/s
内存 (Memory, DRAM)	10~100GB	\$10/GB	50~100ns	~10GB/s
固态硬盘 (SSD, Flash)	100~1TB	\$0.5~\$1/GB	读~100us 写100~1ms	300MB~1GB/s
硬盘 (HDD)	10~100TB	\$0.05/GB	~10ms	~100MB/s

# 一些名词

- Secondary storage: 是指硬盘、固态硬盘等外存
  - 采用标准的块操作接口
  - 通过SATA、SAS、PCI-E等连接到计算机
- Tertiary storage: 通常是大容量的归档式存储
  - 例如：磁带，光盘等
  - 读取数据需要花数秒~数分钟时间
  - 但是可以提供更廉价的存储
- 易失（Volatile）：断电后，存储内容消失
  - 例如：Register, SRAM, DRAM
- 非易失（Non-Volatile）：断电后，存储内容继续保持
  - 例如：磁介质、光介质、闪存等

# 存储层次间传输数据

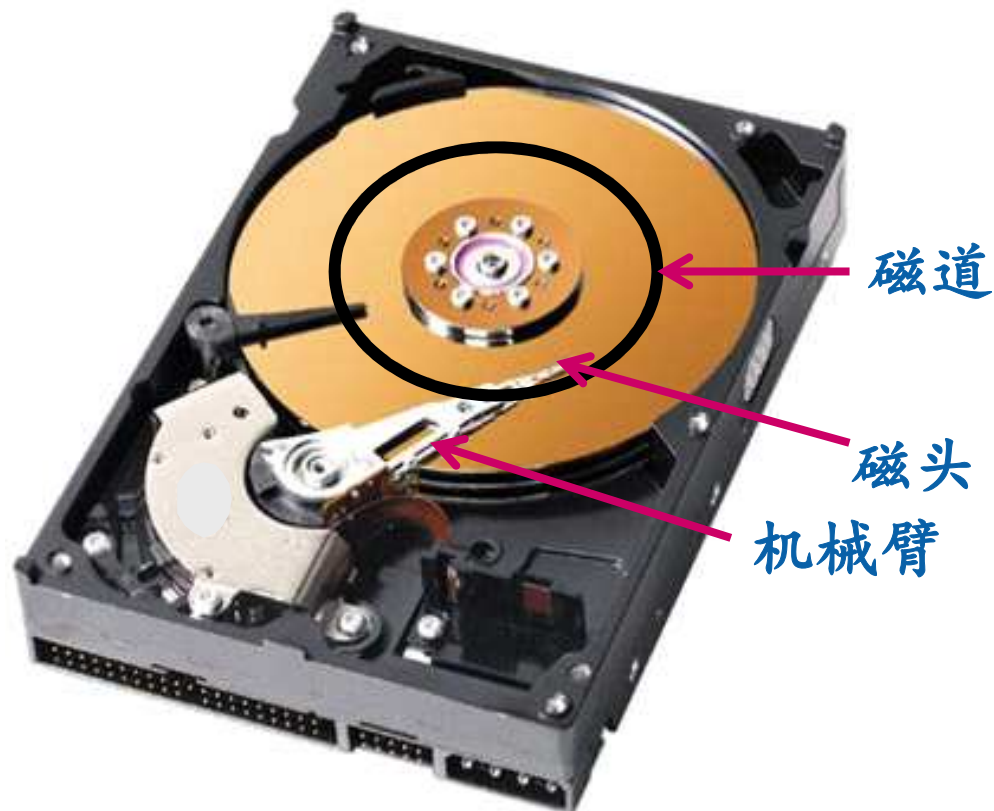




# Outline

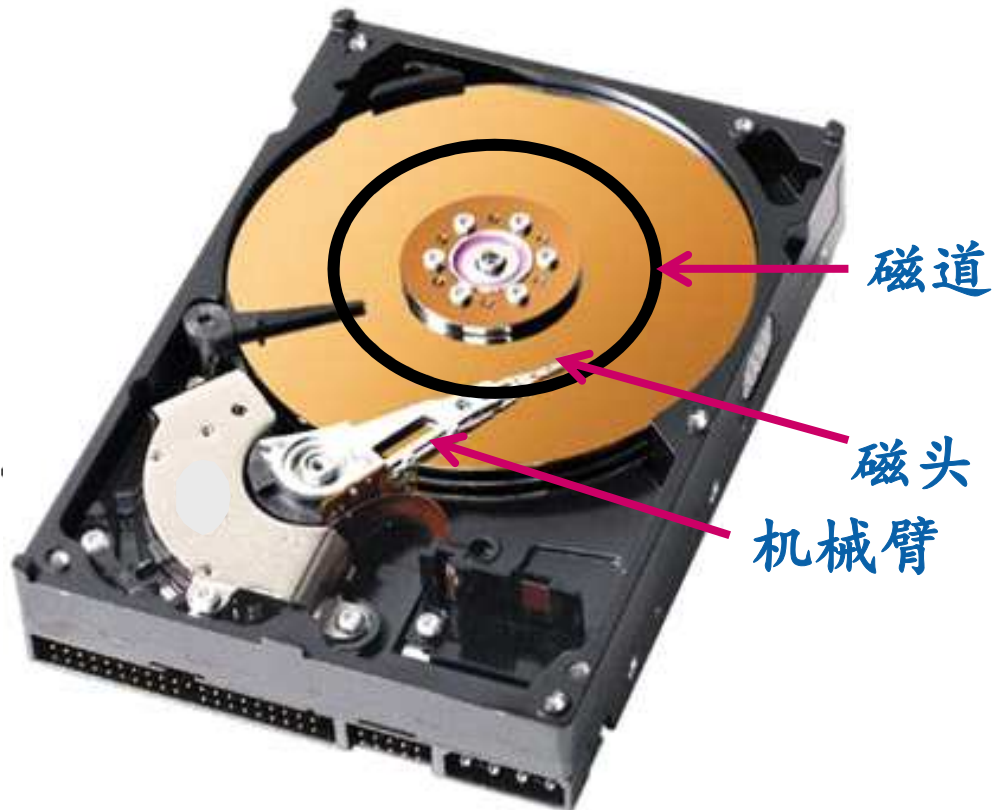
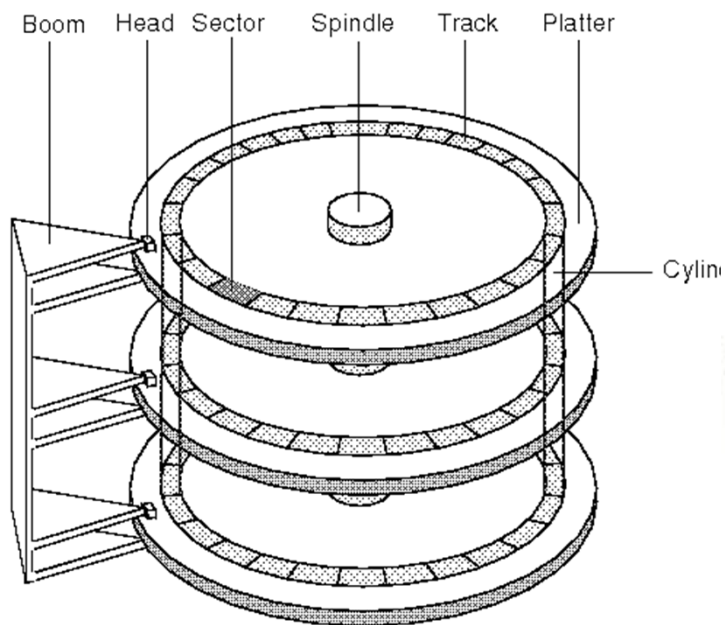
- 数据库系统内部架构概述
- 数据存储与访问路径概述
  - 存储层次
  - 存储介质：磁盘、固态硬盘等
  - 磁盘阵列
  - 操作系统支持
  - 数据与索引
- 磁盘空间管理
- 记录文件格式
- 缓冲区管理

# 硬盘 (HDD, Hard Disk Drive)



- 盘片(Platter)是磁存储介质，通常1~3片
  - 直径5吋，3.5吋（台式机），2.5吋（笔记本/刀片机）
- 正常工作时，盘片匀速旋转(spin), rpm (round per minute)
  - 5400rpm (笔记本), 7200rpm (台式机), 10000rpm (服务器), 15000rpm

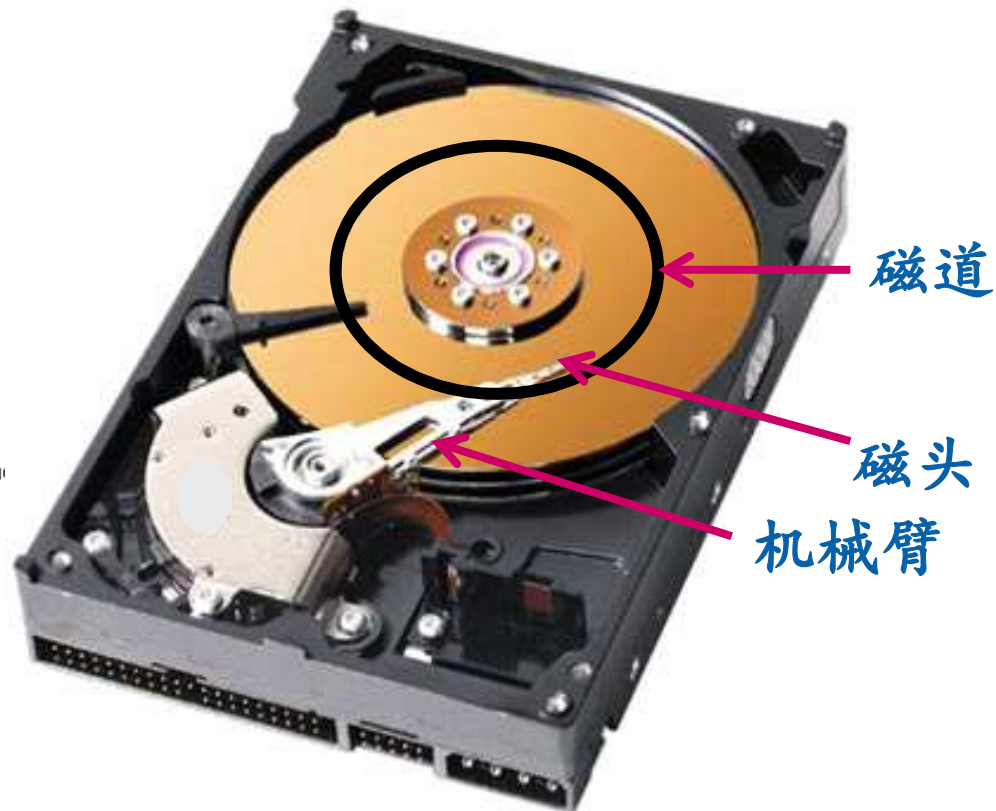
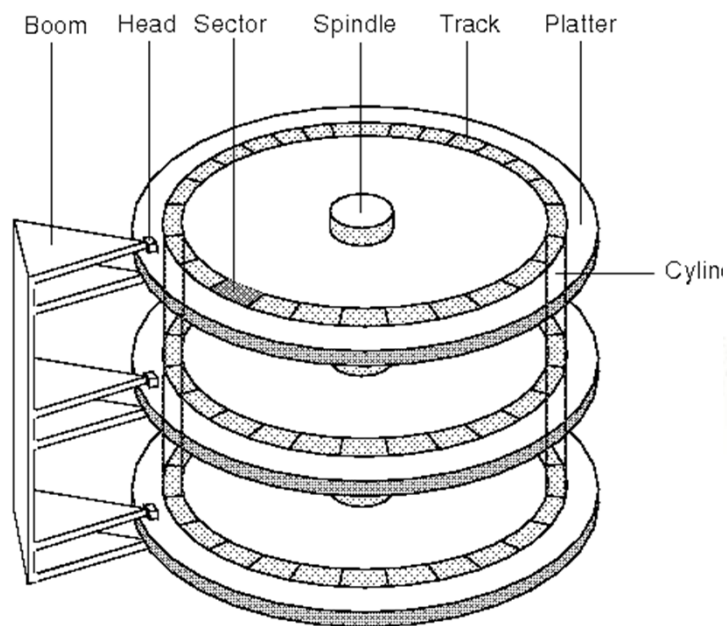
# 硬盘 (HDD)



- 盘片(Platter)分成许多的同心圆

- 每个同心圆称作一个磁道 (track), (2008年:  $10^5$  磁道/盘)
- 多个盘片同一位置的磁道称作一个柱面 (cylinder)
- 每个磁道进一步分成许多扇区 (sector)
- 每个扇区可以存储512B数据

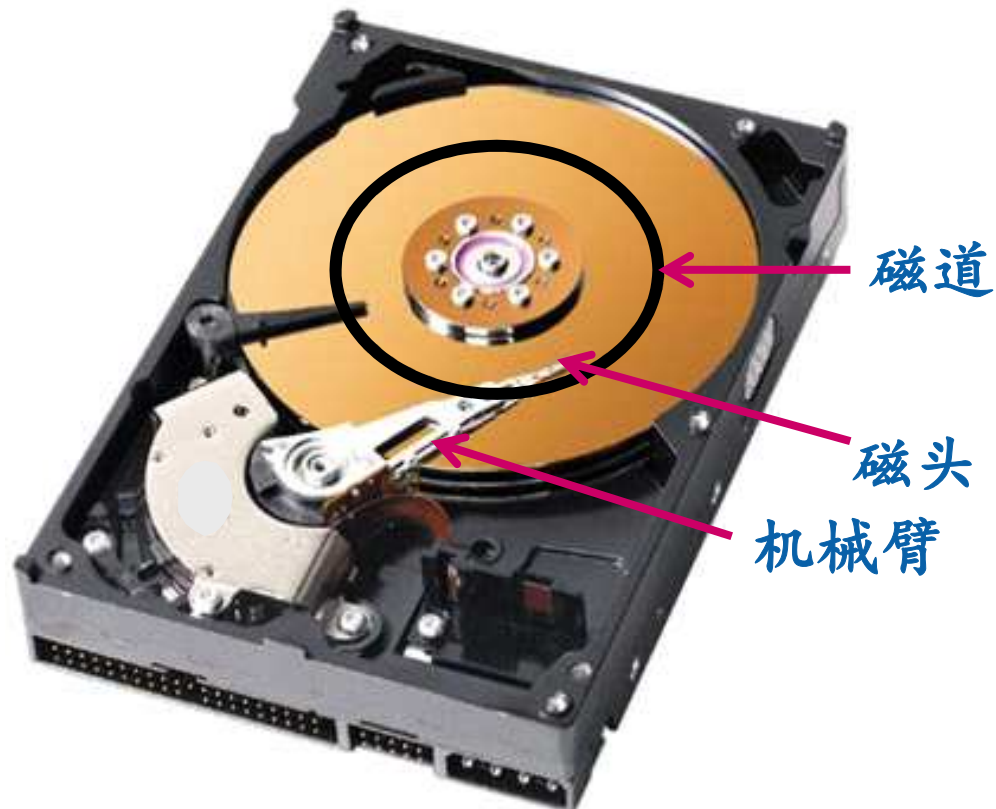
# 硬盘 (HDD)



## • 磁头

- ❑ 盘片高速旋转，在磁头和盘片之间形成一个空气薄膜
- ❑ 磁头悬浮在盘片上
- ❑ 如果磁头和盘片接触，就会破坏盘面
  - 例如，突然断电，或者震动

# 硬盘访问



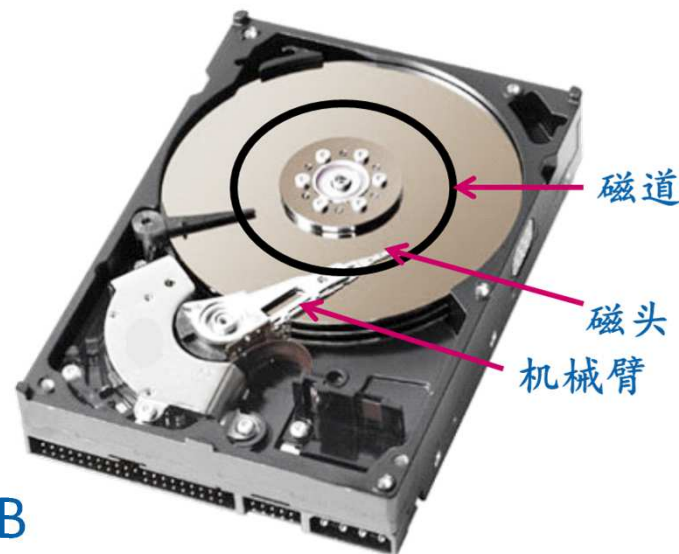
- 给定一个具体的track和sector，访问过程如下

- Seek: 移动磁头，放置到相应的cylinder上
- Rotational delay: 等待sector旋转到磁头之下
- Transfer: 读/写sector的内容

- 访问时间 =  $T_{\text{seek}} + T_{\text{rotate}} + T_{\text{transfer}}$

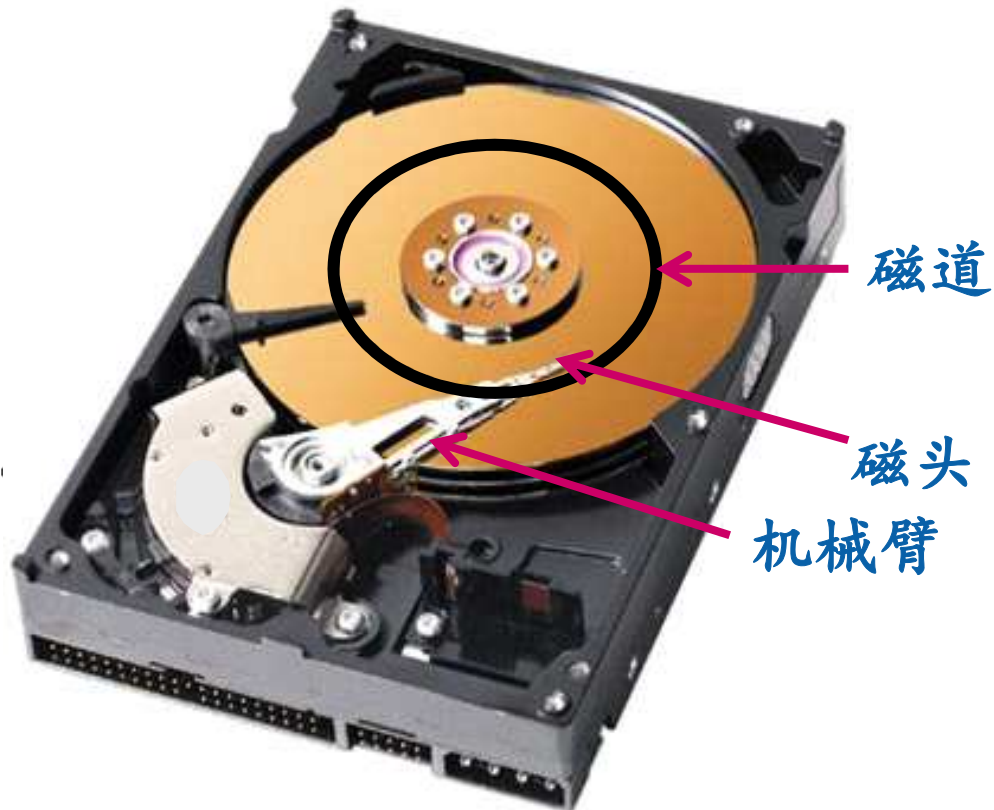
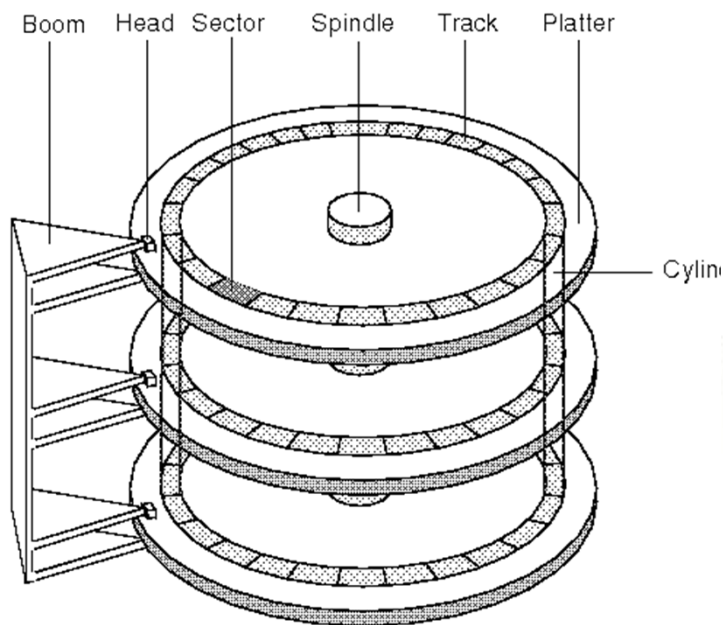


# 随机访问vs.顺序访问



- 访问时间 =  $T_{\text{seek}} + T_{\text{rotate}} + T_{\text{transfer}}$
- 随机访问：读取很少量的数据，例如4KB
  - $T_{\text{seek}} + T_{\text{rotate}}$  起主要作用
  - 每次随机访问~10ms，每秒进行100次访问
  - 例如：随机访问4KB数据，那么400KB/s
- 顺序访问：读大量的数据，例如>1MB
  - $T_{\text{transfer}}$  起主要作用
  - 速度取决于盘片转速、磁介质密度、硬盘接口带宽限制
  - ~100MB/s
- 优化目标：尽量使用顺序访问

# 硬盘 (HDD)

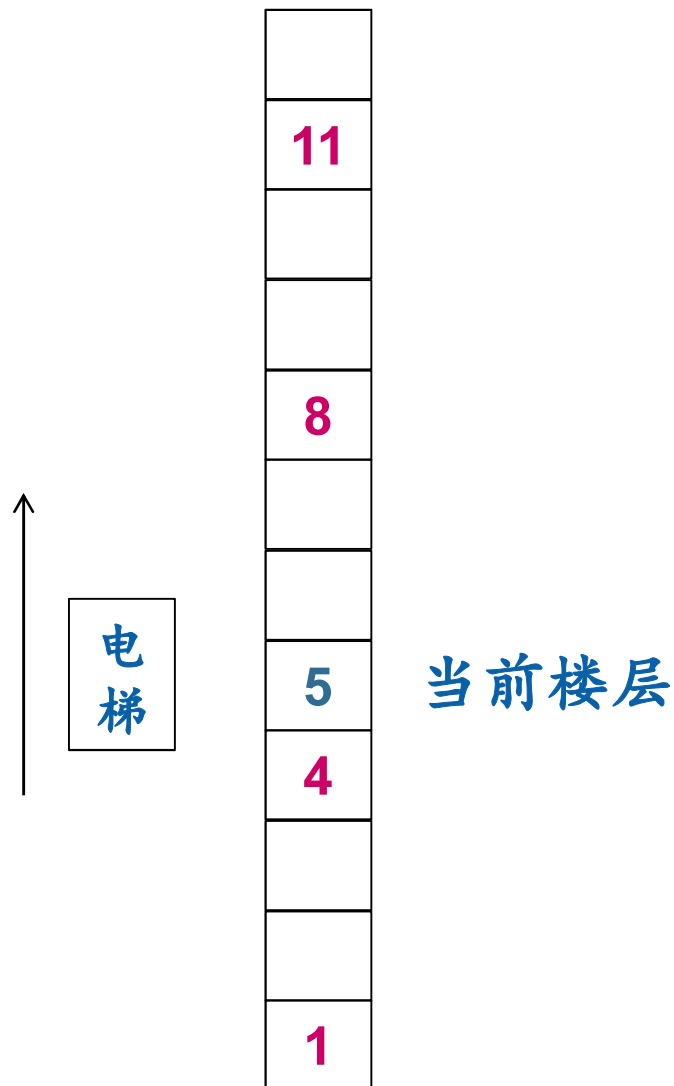


## • 磁盘控制器

- 控制机械臂的移动，完成读写
- 实现外设接口协议：例如SATA, SCSI, 或PCI
- 通常含有一定的缓冲区（10~100MB大小）
- 实现了磁盘请求调度（对于一组请求按什么顺序完成？）

# 电梯调度 (Elevator Algorithm)

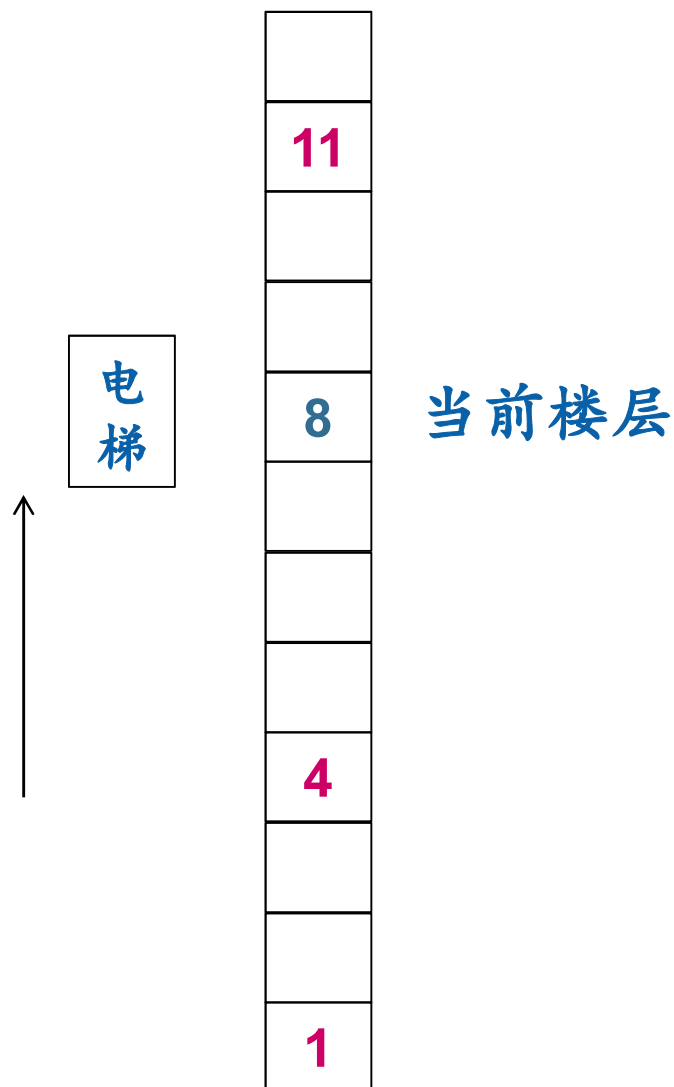
- 一种简单常见的算法
- 电梯是如何运行的？





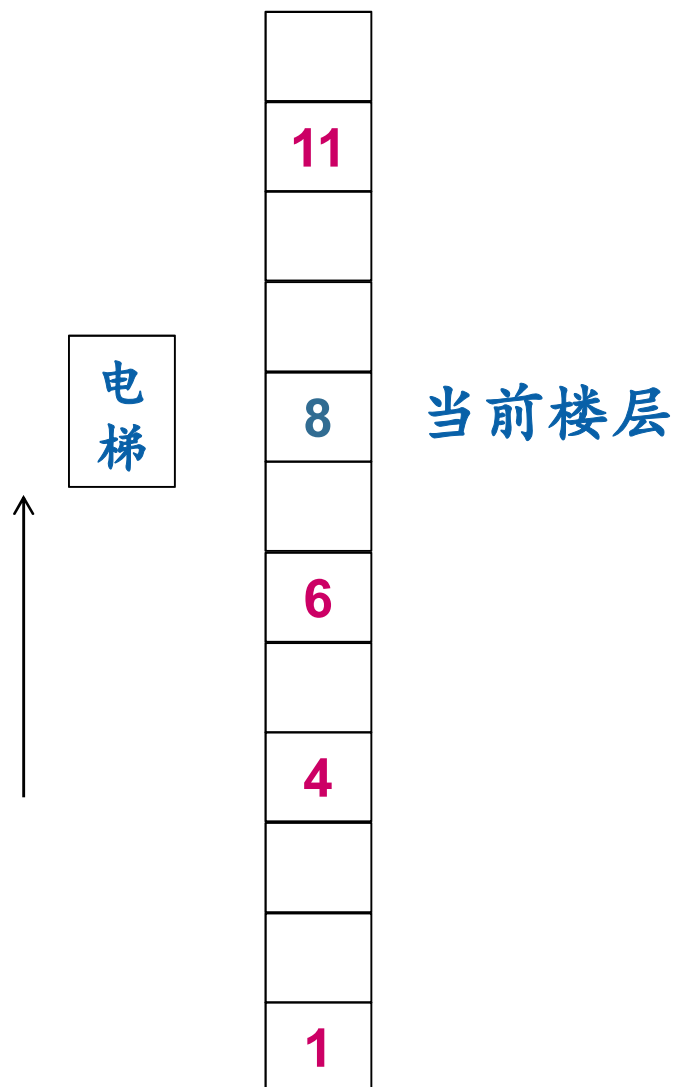
# 电梯调度 (Elevator Algorithm)

- 一种简单常见的算法
- 电梯是如何运行的？



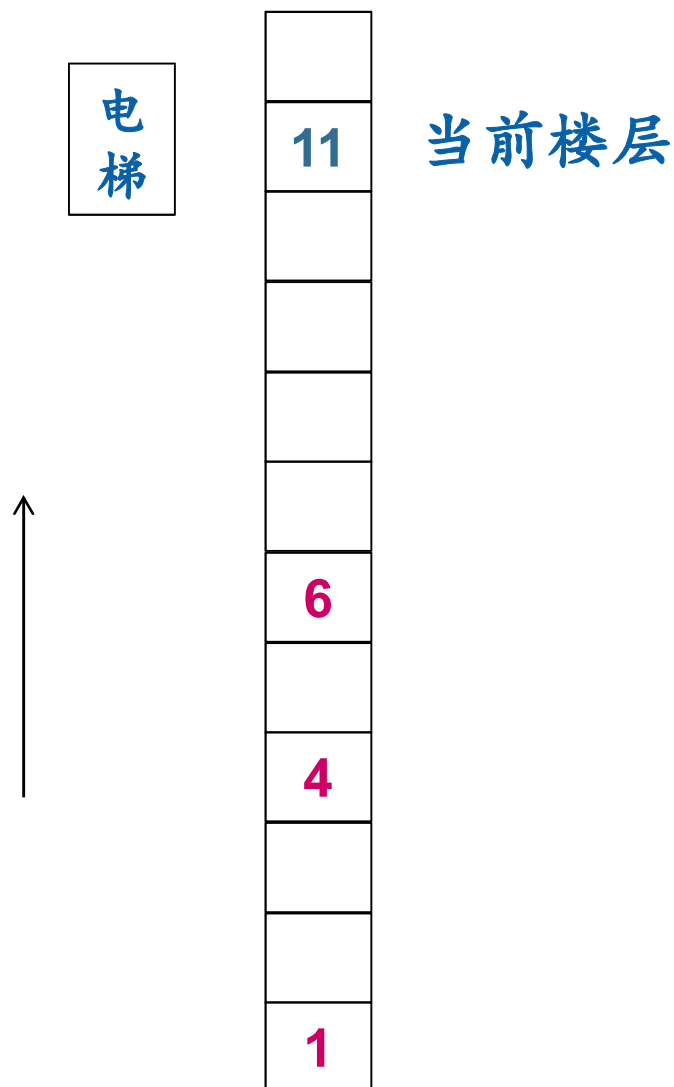
# 电梯调度 (Elevator Algorithm)

- 一种简单常见的算法
- 电梯是如何运行的？



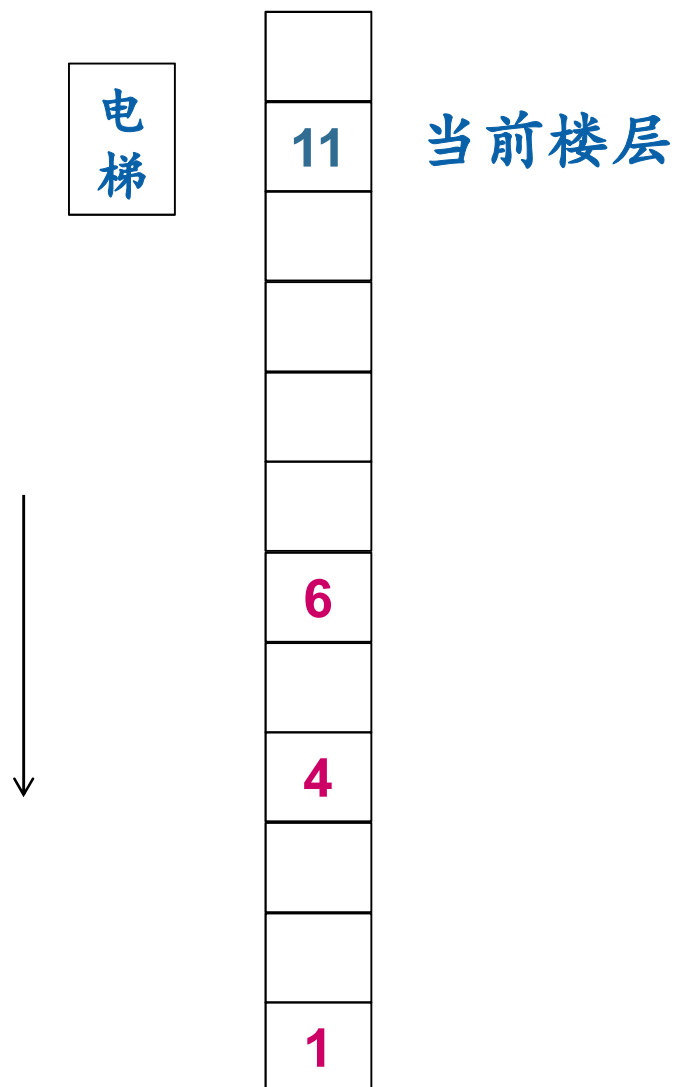
# 电梯调度 (Elevator Algorithm)

- 一种简单常见的算法
- 电梯是如何运行的？



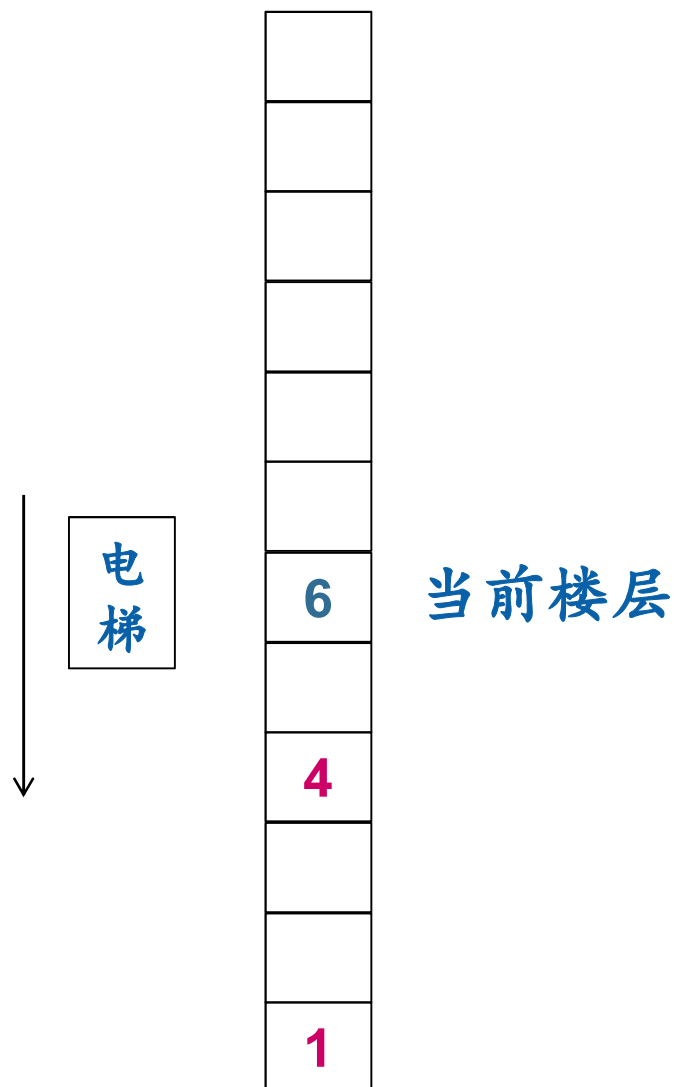
# 电梯调度 (Elevator Algorithm)

- 一种简单常见的算法
- 电梯是如何运行的？



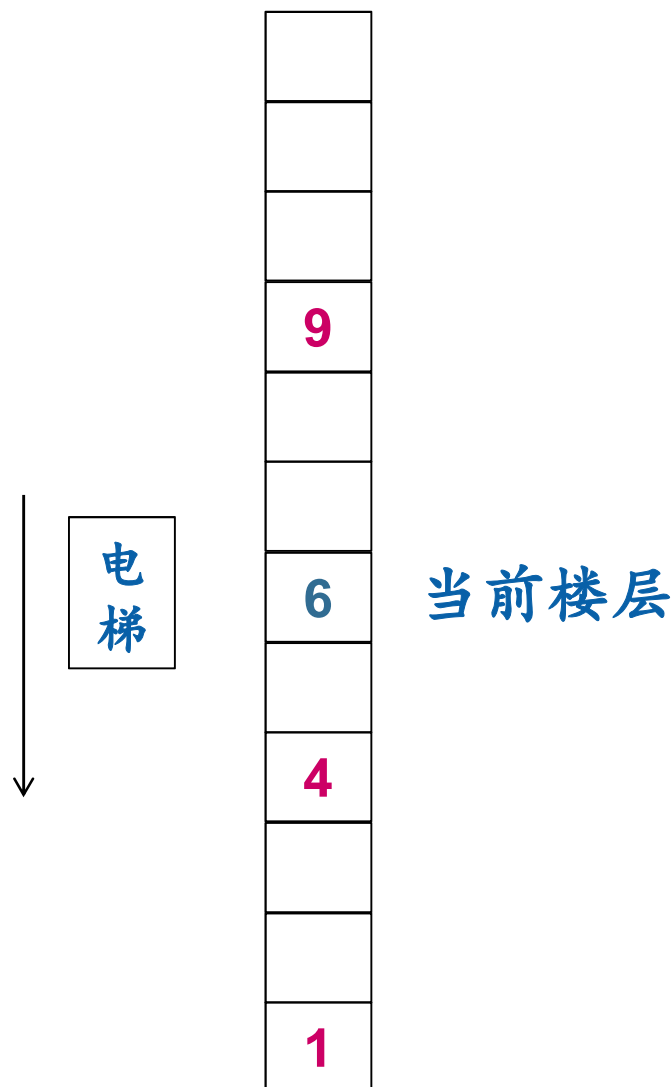
# 电梯调度 (Elevator Algorithm)

- 一种简单常见的算法
- 电梯是如何运行的？



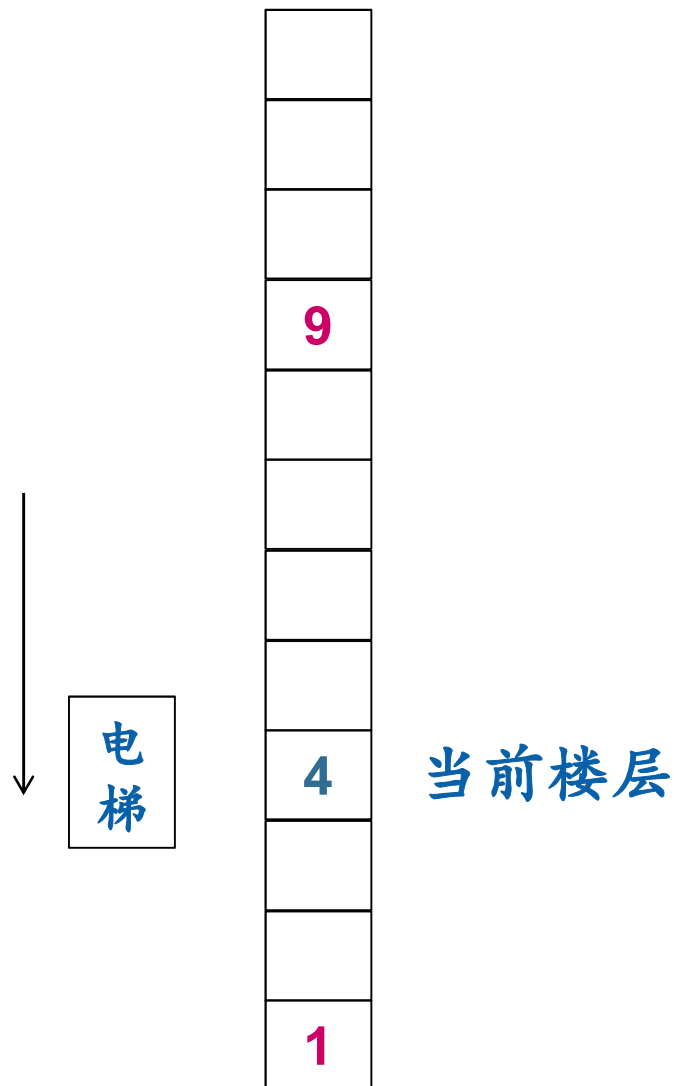
# 电梯调度 (Elevator Algorithm)

- 一种简单常见的算法
- 电梯是如何运行的？



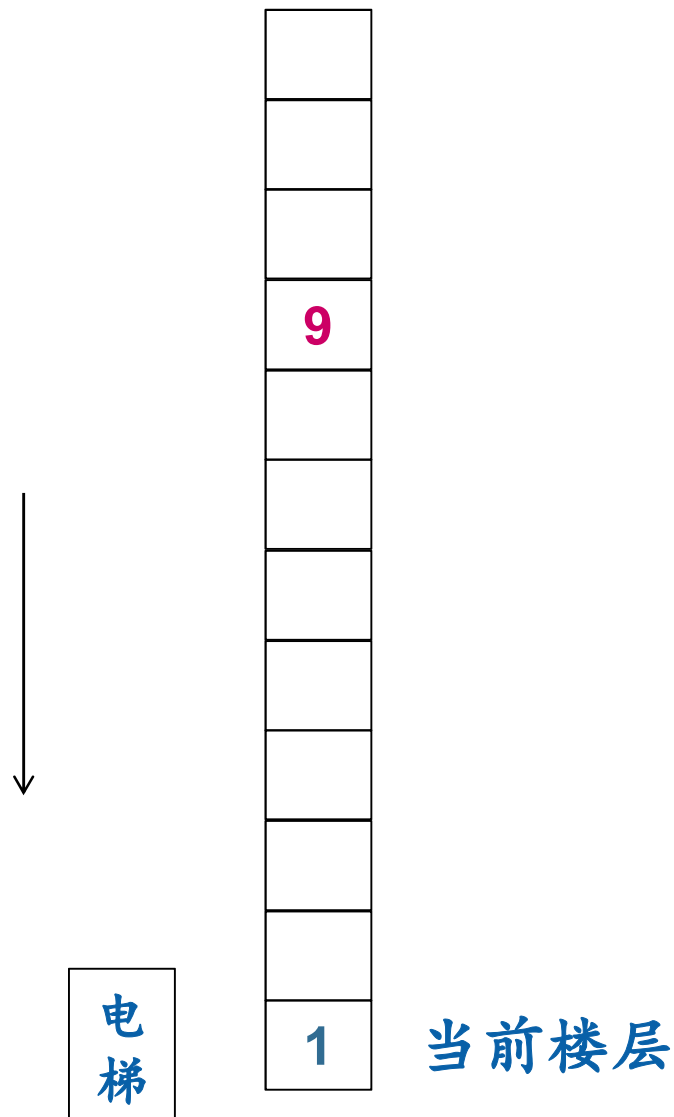
# 电梯调度 (Elevator Algorithm)

- 一种简单常见的算法
- 电梯是如何运行的？



# 电梯调度 (Elevator Algorithm)

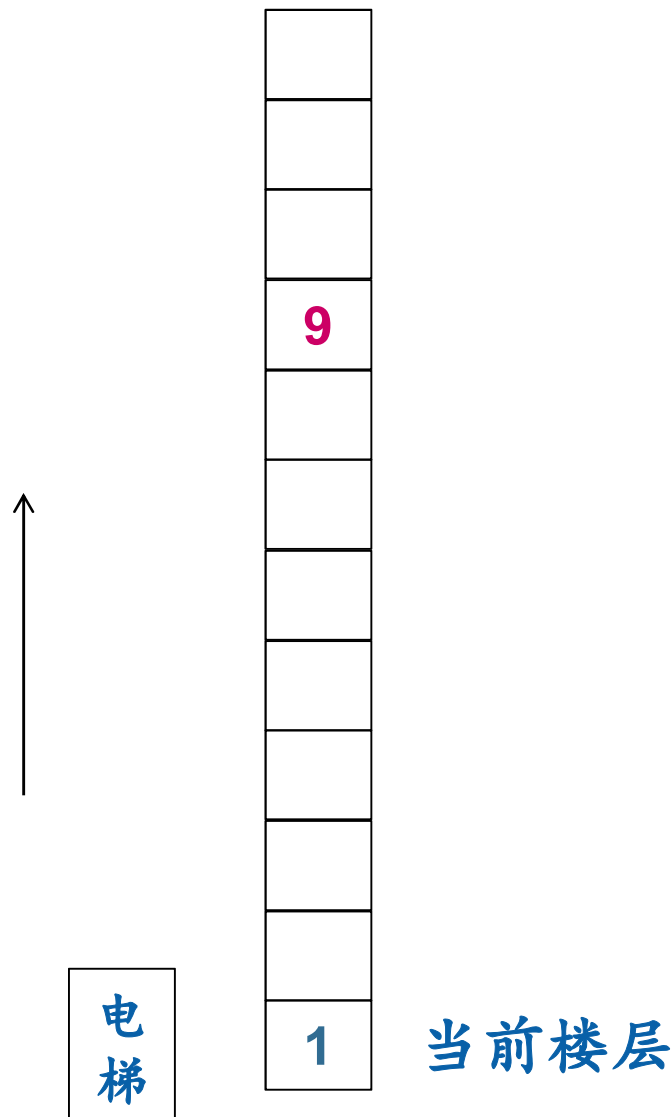
- 一种简单常见的算法
- 电梯是如何运行的？





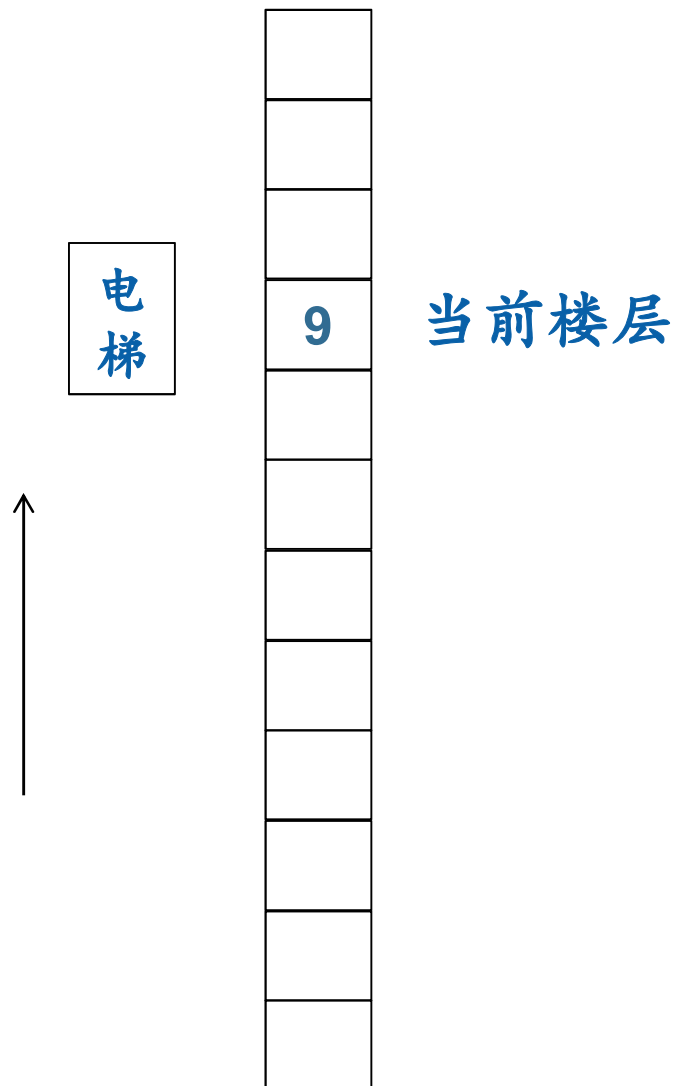
# 电梯调度 (Elevator Algorithm)

- 一种简单常见的算法
- 电梯是如何运行的？



# 电梯调度 (Elevator Algorithm)

- 一种简单常见的算法
- 电梯是如何运行的？



# 电梯调度 (Elevator Algorithm)

```
request = {...}; direction= forward; head= 0;
while (request !=  $\emptyset$ ) {
    if (direction == forward) {
        selected= max_int;
        foreach r  $\in$  request { //比head大的最小请求
            if (r > head) selected= min(selected, r);
        }
        if (selected != max_int) {
            seek_and_visit; head= selected;
        } else {
            direction= backward;
        }
    } else {
        // direction == backward, 类似处理
        // . . . . .
    }
}
```

# 闪存(Flash)与固态硬盘(SSD: Solid State Drive)

## • 闪存

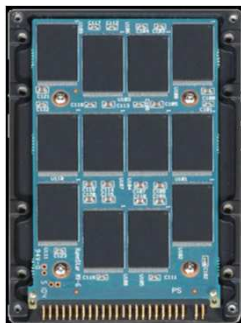
- 发明于1980年，与DRAM技术有一定的相似性
- 最早用于取代ROM作BIOS存储
- 后来用于数字电子设备：相机、手机、mp3、U盘、microSD卡等，大量生产，价格降低

## • 固态硬盘

- 2009年开始出现以闪存为存储介质的固态硬盘



SATA

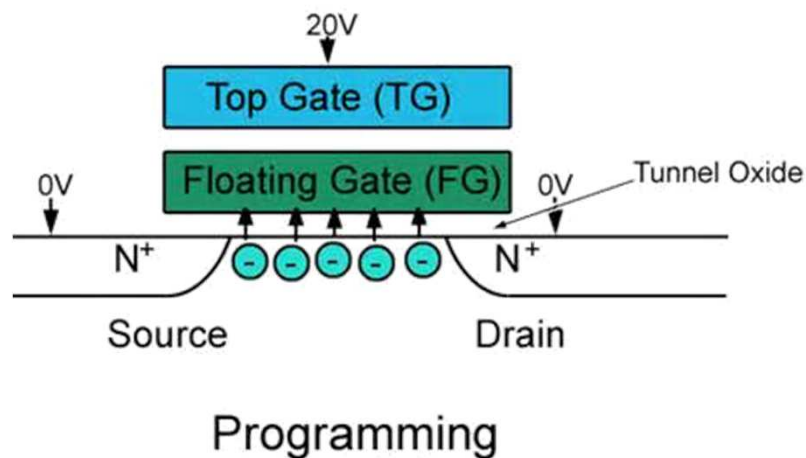


PCI-E

# Flash

- Flash(闪存)的单元结构与DRAM(内存)相似
  - DRAM: 电容存储电荷, 表示0/1
  - Flash: 用一个Floating Gate(浮栅)结构, 保持电荷

## NAND Flash Cell



# 闪存(Flash)读写操作



- 读操作

- 相对简单

- 写操作

- 无法直接修改
  - 擦除(Erase): 对大块数据操作 (例如~1MB), 设为同一状态 (例如 全1)
  - 写(Write): 对一页 (例如4KB) 进行操作, 只能单向修改 (例如1->0)

- FTL: Flash Translation Layer

- 把逻辑地址映射为内部的物理地址
  - 实际上, 写一个逻辑地址是写到了另一个空闲块
  - 内部有比较复杂的管理算法

# 闪存(Flash)读写性能

- 随机读操作，性能很好
  - $\sim 10000$ 次/s
- 随机写操作，性能差
  - 而且会使SSD总体性能变差
- 顺序读写，可以利用内部多个Flash芯片的并行性
  - $300\text{MB} \sim 1\text{GB/s}$



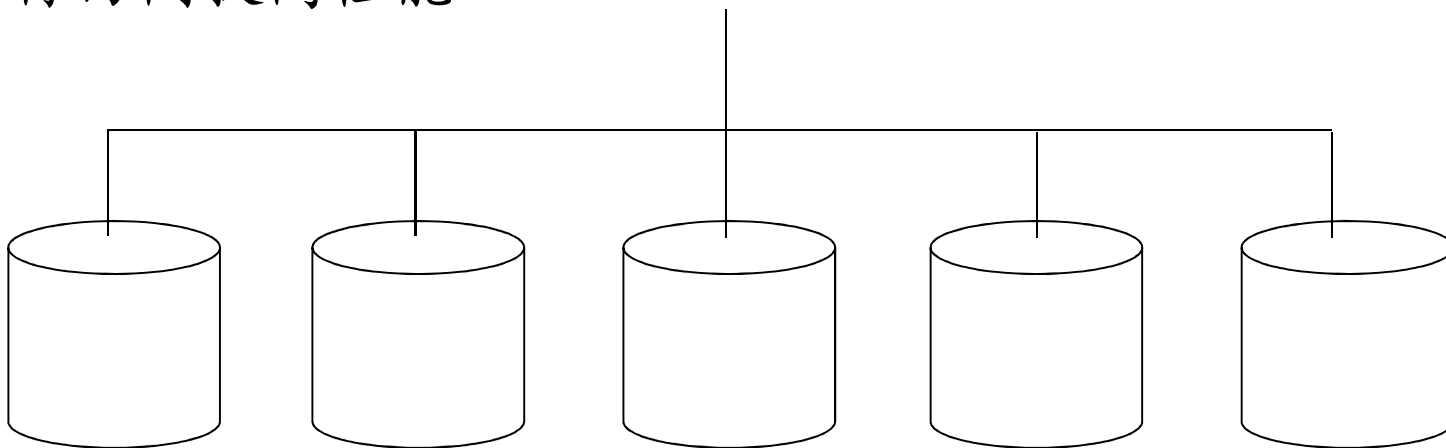
# Outline

- 数据库系统内部架构概述
- 数据存储与访问路径概述
  - 存储层次
  - 存储介质：磁盘、固态硬盘等
  - 磁盘阵列
  - 操作系统支持
  - 数据与索引
- 磁盘空间管理
- 记录文件格式
- 缓冲区管理



# 磁盘阵列 (Disk Array)

- 用多个磁盘组成一个磁盘阵列
- 对外提供统一的一个块操作界面
- 内部通过多个磁盘实现
  - 数据冗余+容错
  - 并行访问提高性能



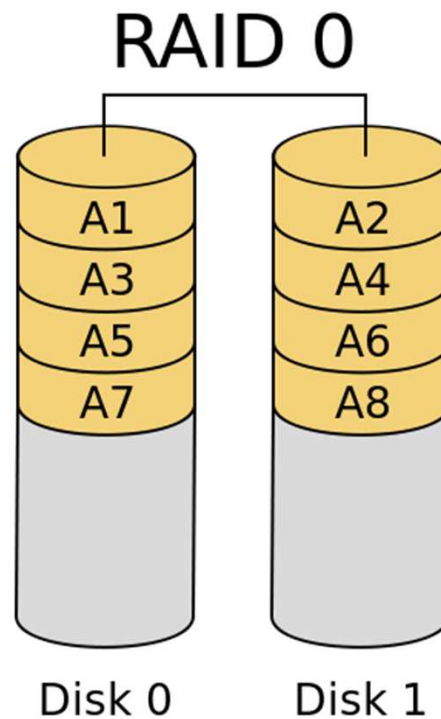
# RAID

- Redundant Array of Independent Disks
- RAID levels
  - RAID 0
  - RAID 1
  - RAID 5
  - RAID 0+1
  - RAID 10 (即 RAID 1+0)
  - RAID 50 (即 RAID 5+0)

# RAID 0

## • Striping

- stripe unit: 例如64KB
- 想象所有磁盘空间都分解为64KB大小的对齐的块
- 那么，全局的第N块
  - 在第  $N \% \text{num\_disks}$  个磁盘上  
(磁盘从0号开始)
  - 在磁盘内部是第  $N / \text{num\_disks}$  块  
(块从第0块开始)
- 例如:  $\text{num\_disks} = 2$
- 那么全局第 103 块在哪里?

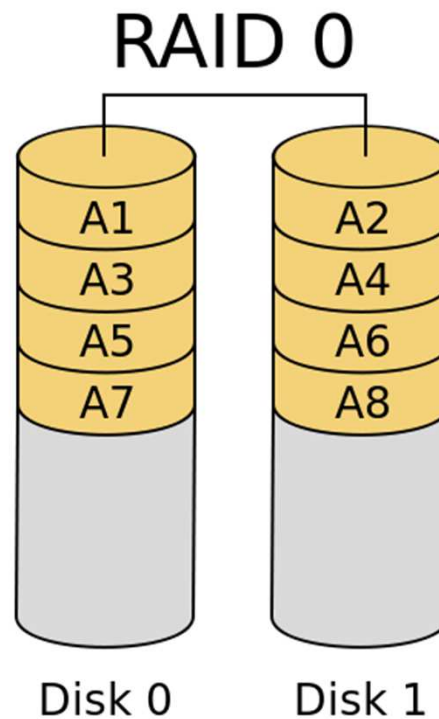


图像来源: wikipedia

# RAID 0

## • Striping

- stripe unit: 例如64KB
- 想象所有磁盘空间都分解为64KB大小的对齐的块
- 那么，全局的第N块
  - 在第  $N \% \text{num\_disks}$  个磁盘上  
(磁盘从0号开始)
  - 在磁盘内部是第  $N / \text{num\_disks}$  块  
(块从第0块开始)
- 例如:  $\text{num\_disks} = 2$
- 那么全局第 103 块在哪里?
  - $103 \% 2 = 1$ , 在磁盘1上
  - $103 / 2 = 51$ , 在盘内块51的位置



图像来源: wikipedia

# RAID 0

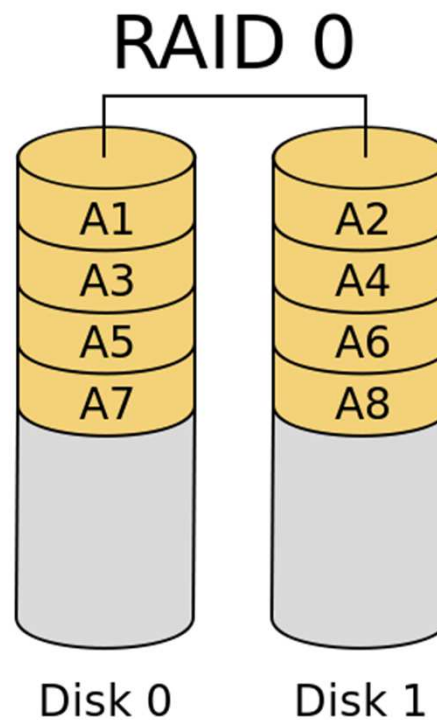
## • Striping

### □ 优点

- 随机访问时：可能同时访问多个盘
- 顺序访问时：发挥多个盘的总带宽

### □ 缺点

- 不提供数据冗余
- 如果磁盘坏了，那么数据丢失



图像来源：wikipedia

# RAID 1

- Mirroring（镜像）

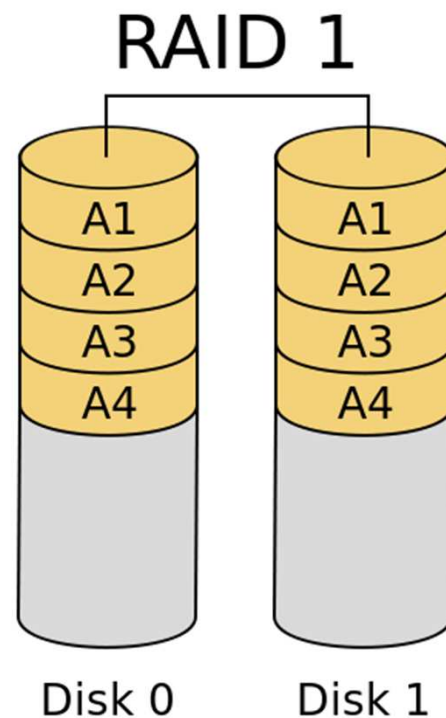
- 两个硬盘完全一样

- 优点

- 数据冗余：而且恢复速度快
    - 读操作：可以并行执行

- 缺点

- 写操作：要写两个盘

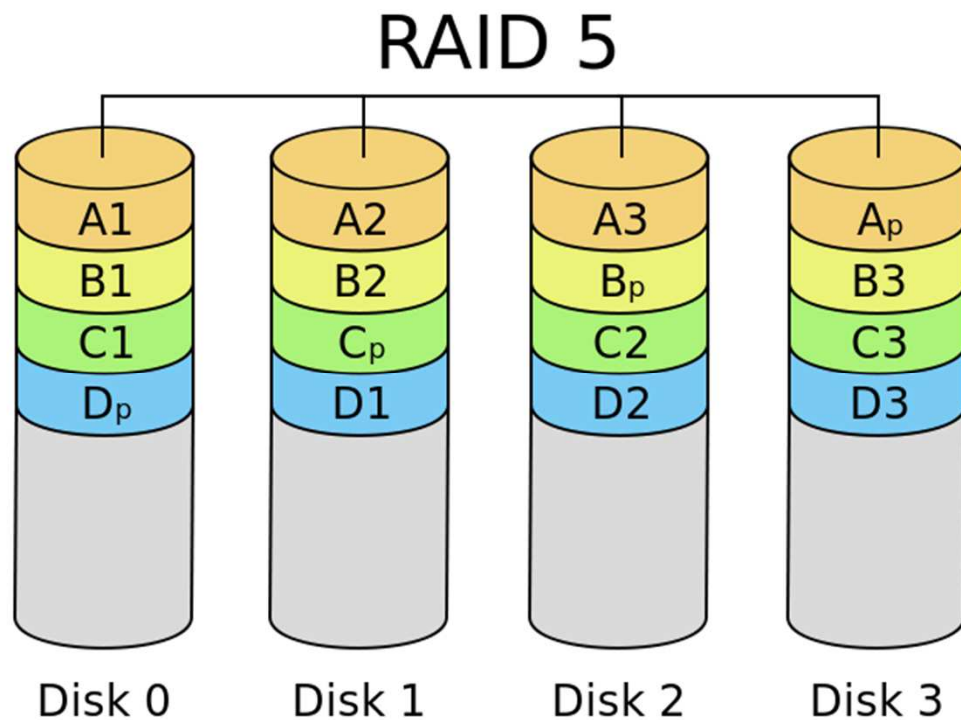


图像来源：wikipedia

# RAID 5

- RAID 5

- 采用Parity（异或）校验一组Stripe Unit
- Parity块轮流放置在不同的硬盘上



图像来源: wikipedia

# Parity（又称为奇偶校验）

- 盘1: 0001
- 盘2: 0110
- 盘3: 1001
- Parity: 1110（奇数个1为1，偶数个1为0）

- 假设盘3坏了，那么可以通过计算得到

□  $0001 \oplus 0110 \oplus 1110 = 1001$

□ 于是就可以恢复盘3的数据



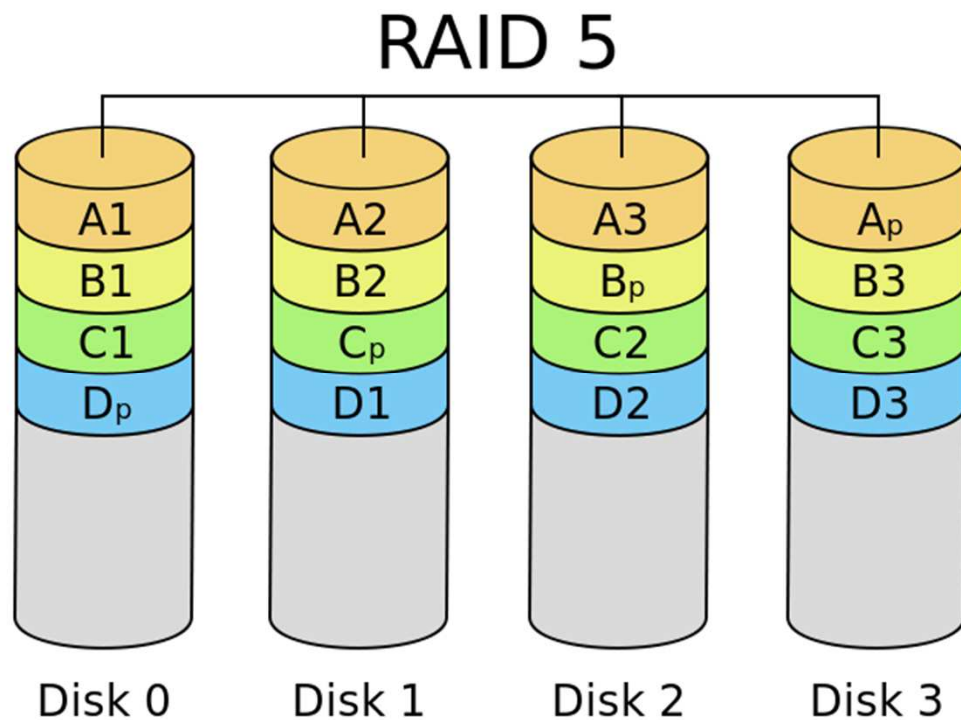
# RAID 5

## • 优点

- 校验和冗余：可以处理一个磁盘损坏的情况
- 读操作：可以发挥并行性

## • 缺点

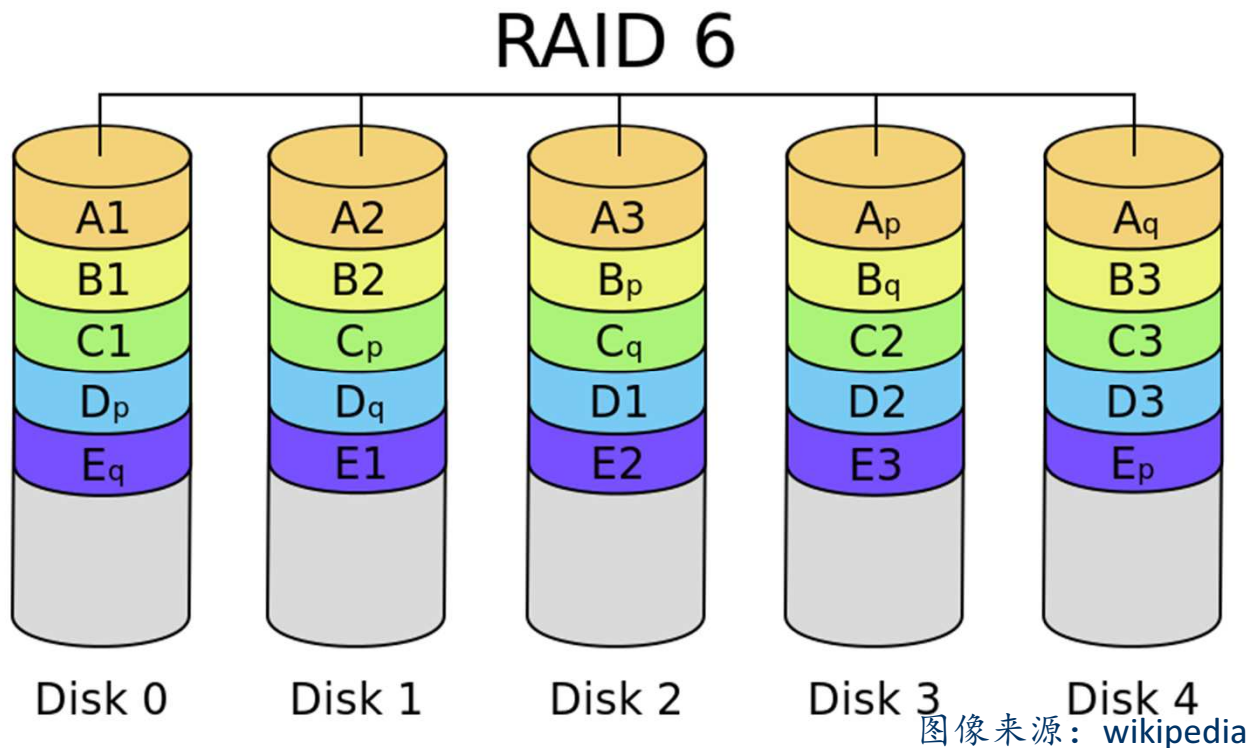
- 写操作要重新计算Parity
  - 写原始块和Parity块
- 恢复时，要读所有的磁盘才能恢复，代价大



图像来源: wikipedia

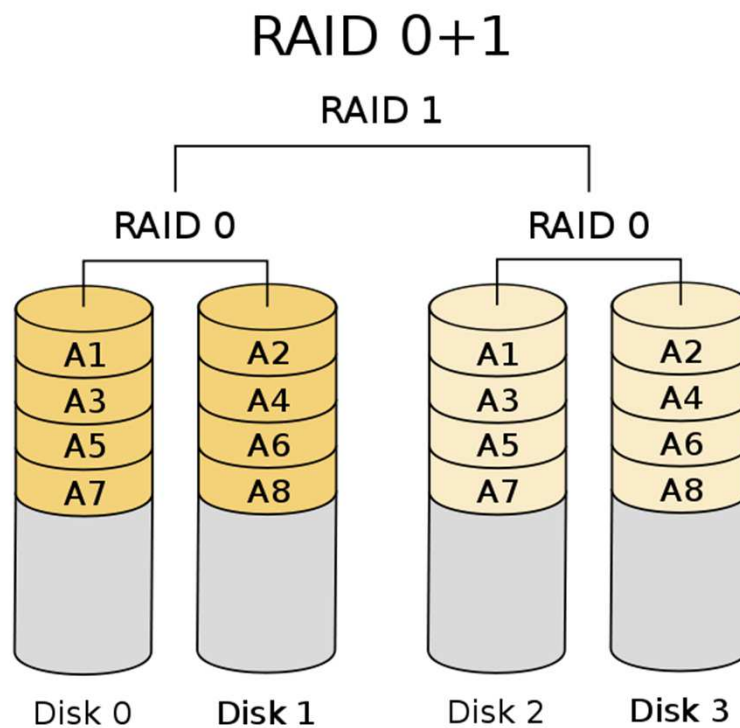
# RAID 6

- 与RAID5相似，只是每组strip units产生了多个Parity块
  - 所以可以修复多块磁盘损坏的情况
- $k$ 个Parity块最多可以支持 $2^k - 1 - k$ 块磁盘，容忍 $k - 1$ 个坏盘



# RAID 0+1

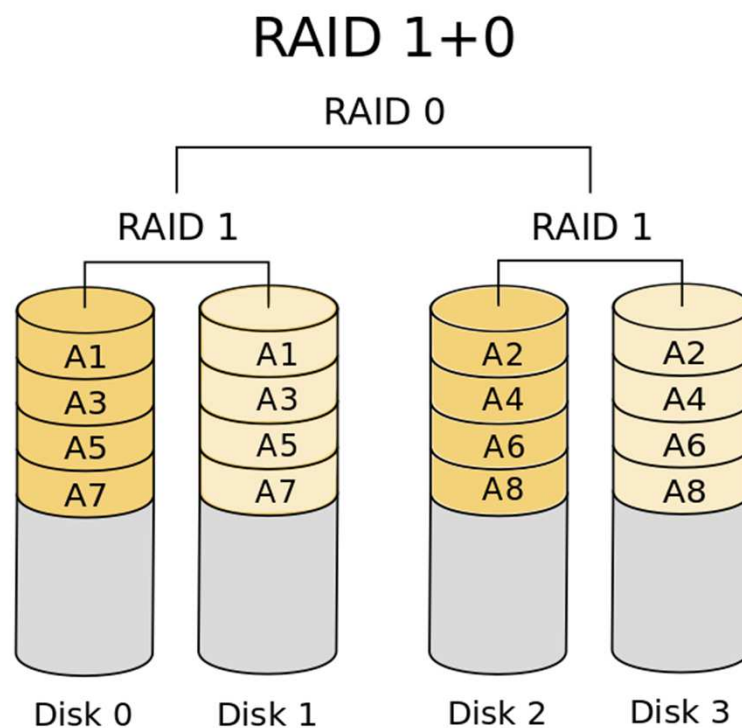
- 下层是RAID0，上面是RAID1



图像来源: wikipedia

# RAID 1+0

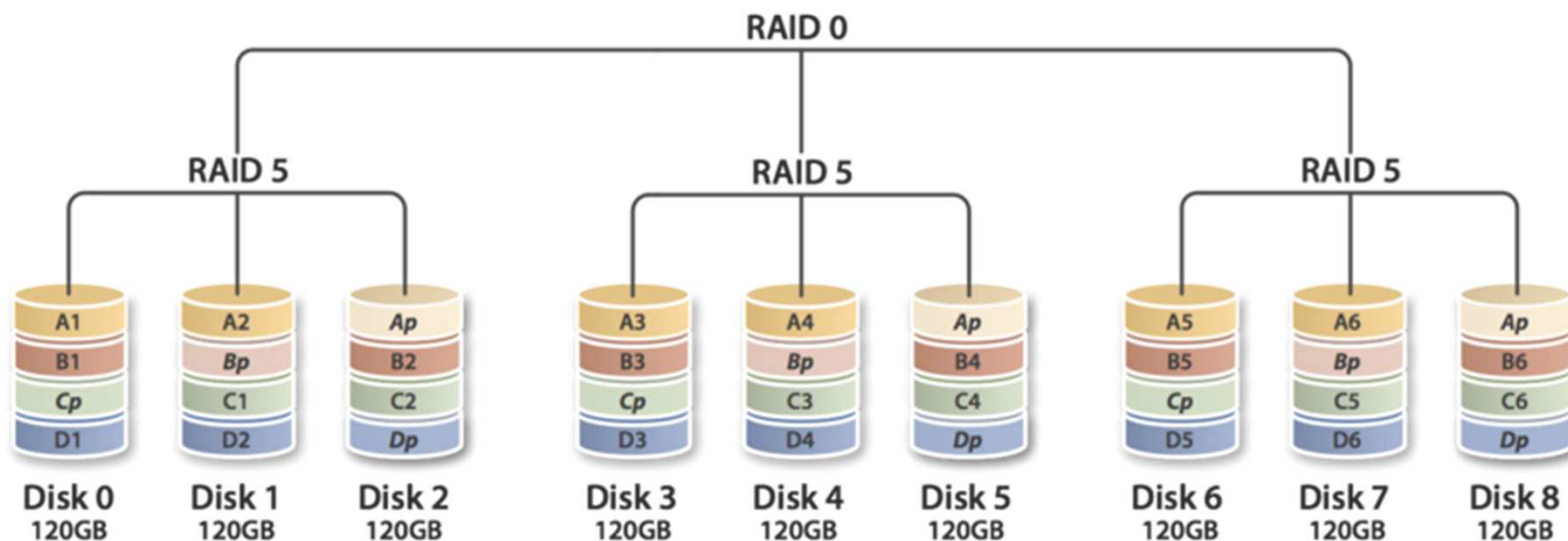
- 下层是RAID1，上面是RAID0



图像来源: wikipedia

# RAID 5+0

- 下层是RAID5，上面是RAID0



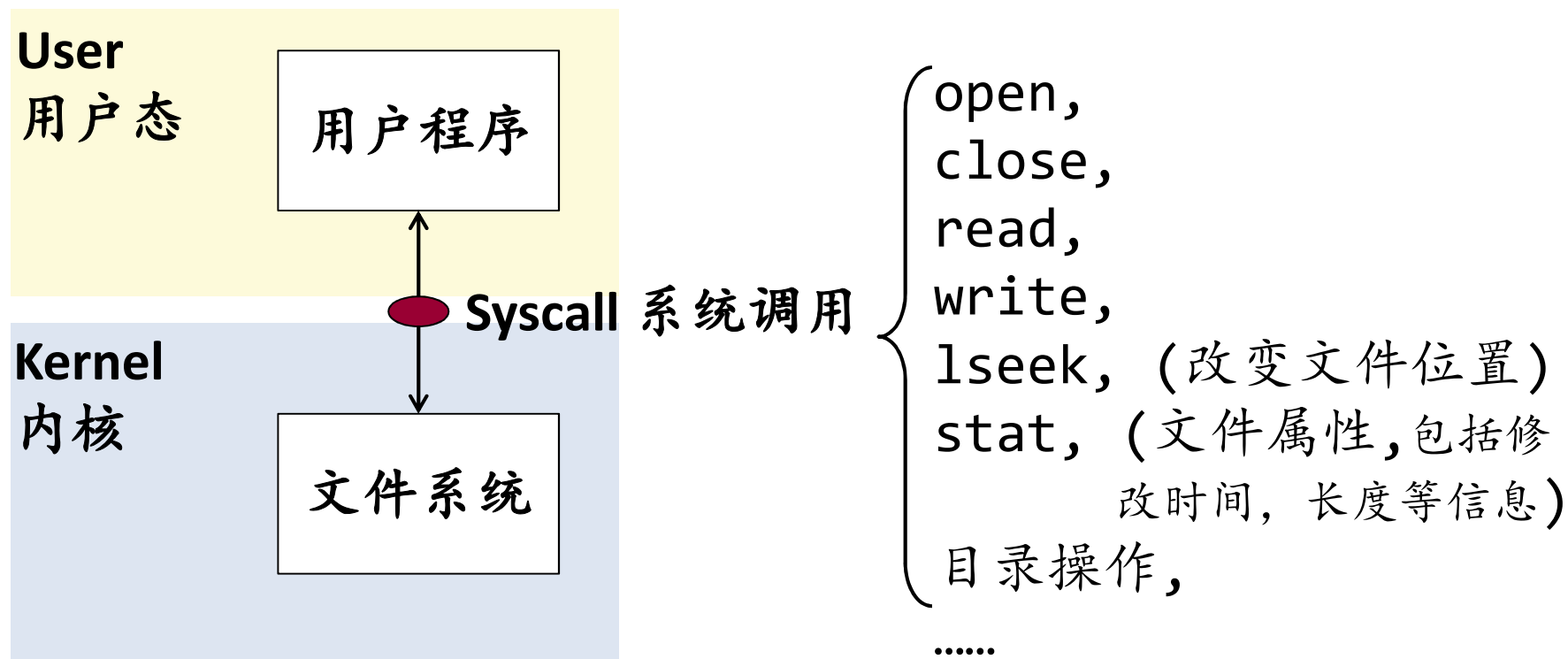
图像来源: wikipedia

# Outline

- 数据库系统内部架构概述
- 数据存储与访问路径概述
  - 存储层次
  - 存储介质：磁盘、固态硬盘等
  - 磁盘阵列
  - 操作系统支持
  - 数据与索引
- 磁盘空间管理
- 记录文件格式
- 缓冲区管理

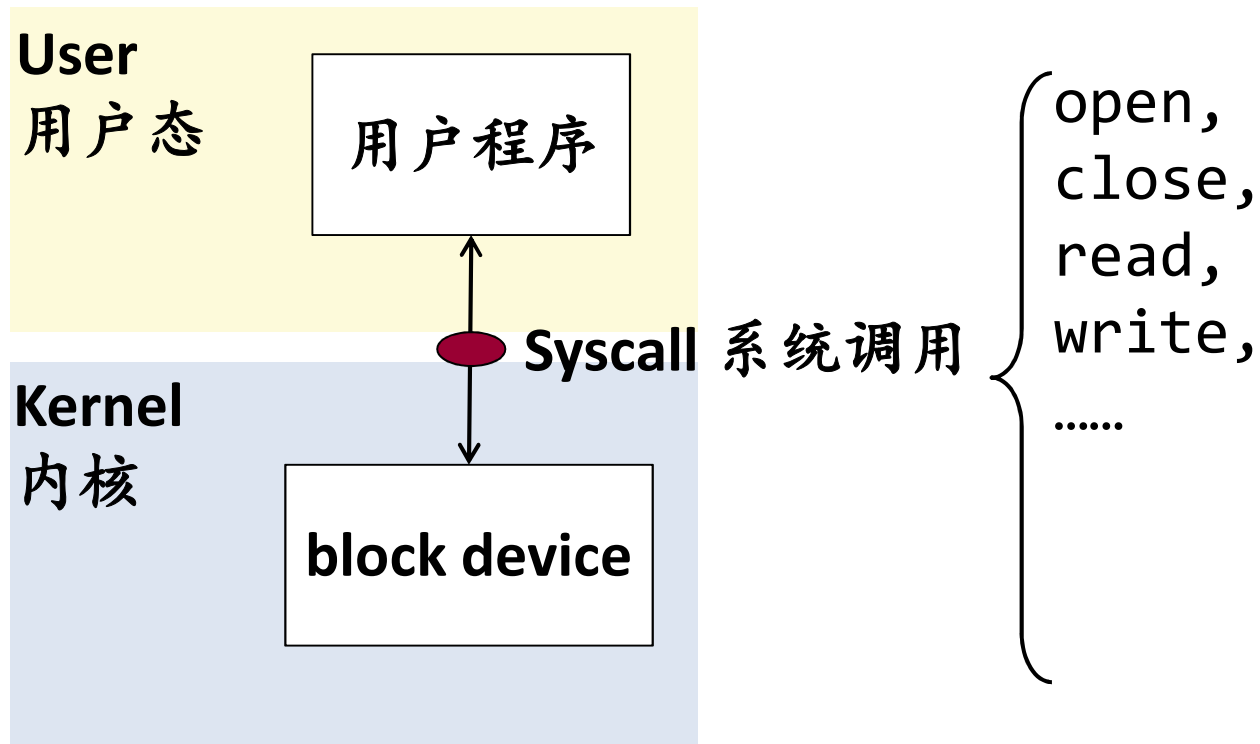
# 本地文件系统 (Local File System)

- 例子：Linux ext4, Windows ntfs, Mac OS hfs ...



# 块设备

- 可以直接打开Raw Partition





# 数据库 vs. 文件系统（数据存储角度比较）

## • 文件系统

- 存储文件(file)
- 通用的，存储任何数据和程序
- 文件是无结构的，是一串字节组成的
- 操作系统内核中实现
- 提供基本的编程接口
  - Open, close, read, write

## • 共同点

- 数据存储在外存（硬盘）
- 根据硬盘特征，数据分成定长的数据块

## • 数据库

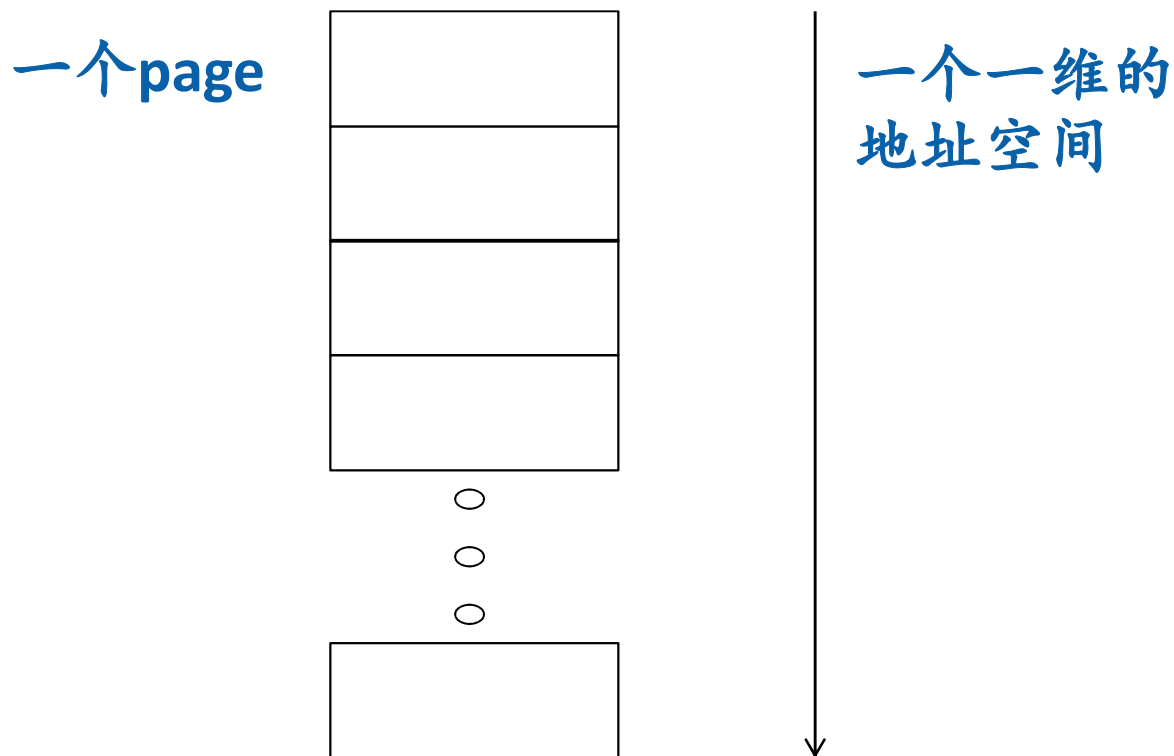
- 存储数据表(table)
- 专用的，针对关系型数据进行存储
- 数据表由记录组成，每个记录由多个属性组成
- 用户态程序中实现
- 提供SQL接口

# 数据在硬盘上的存储

- 硬盘最小存储访问单位为一个扇区：512B
- 文件系统访问硬盘的单位通常为：4KB
- RDBMS最小的存储单位是database page size
  - Data page size 可以设置为1~多个文件系统的 page
  - 例如，4KB, 8KB, 16KB, ...
  - 我们下面用page简称database page



# 数据在硬盘上的存储



- Raw partition或file
- 如果使用file

□ 每个Table可以是自己的文件，也可以使用一个大文件里面分成多个Table

# Outline

- 数据库系统内部架构概述
- 数据存储与访问路径概述
  - 存储层次
  - 存储介质：磁盘、固态硬盘等
  - 磁盘阵列
  - 操作系统支持
  - 数据与索引
- 磁盘空间管理
- 记录文件格式
- 缓冲区管理

# 存储在硬盘上的内容：记录文件

- 记录数据Table

- 行式存储，或者列式存储

- 行式存储的堆文件

- 记录是无序的

- 多条记录组成数据页

- 文件由多个数据页组成

- 排序文件

- 记录按照某个顺序排序

- 支持插入、删除、修改

# 存储在硬盘上的内容：索引

- 索引结构

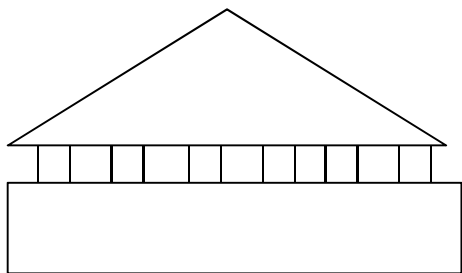
- 树结构索引

- 哈希索引

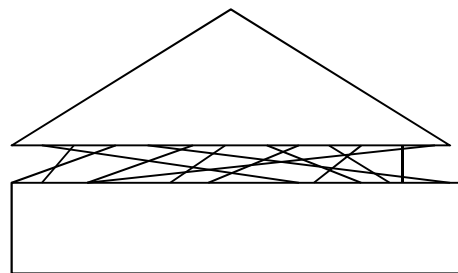
- 聚簇？数据页与索引的顺序一致吗？

- 聚簇索引（Clustered Index）：顺序一致

- 二级索引（Secondary Index）：顺序不一致



**Clustered**



**Non-Clustered**  
又称作**Secondary**

# 存储在硬盘上的内容：其它

- System catalog

- 包含database, table, view, index等的定义
- 包含一些统计信息，用以优化查询

- Log: 日志

- Transactional log: 为了事务处理ACID所使用的log
- Operation log: 每个处理操作的记录

# Outline

- 数据库系统内部架构概述
- 数据存储与访问路径概述
- 硬件和操作系统支持
- 磁盘空间管理
  - 概念
  - 工作原理
- 记录文件格式
- 缓冲区管理



# 比照内存中的malloc/free

- malloc/free

- 通过mmap/sbrk等从操作系统分配新的大块内存
- 把大块内存分解为小的内存块
- 根据应用程序中malloc分配的大小，找到合适的块
- 完成分配与释放

# 在磁盘上需要进行空闲块的管理

- 方法1：数据库自己管理空间

- 下层存储是Raw Partition，磁盘分区
- 或者使用一个大文件，内部自行管理

- 记录数据块是否空闲

- 空闲块列表
- 或者空闲块位图（Bitmap）
  - 0代表空闲，1代表使用

- 完成数据块分配与释放功能

- 进一步，每个Table或Index需要记录所使用的Page

# 在磁盘上需要进行空闲块的管理

- 方法2：依托操作系统的文件系统来管理

- 数据库的每个Table或Index存成单独的文件
- 下层文件系统具体对磁盘的空闲块进行管理

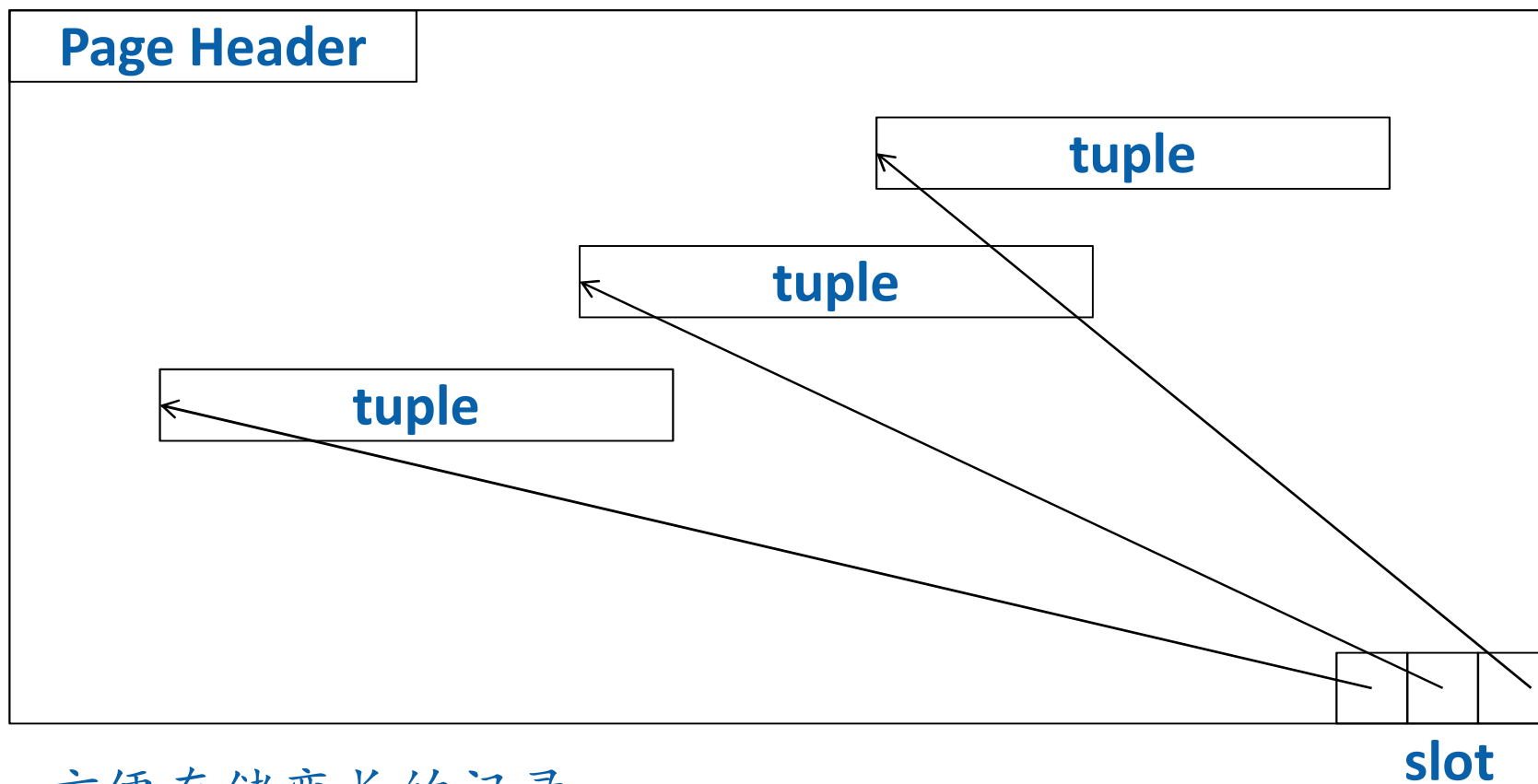
- 现在用得比较多

- 因为文件系统已经相当稳定，性能也很高

# Outline

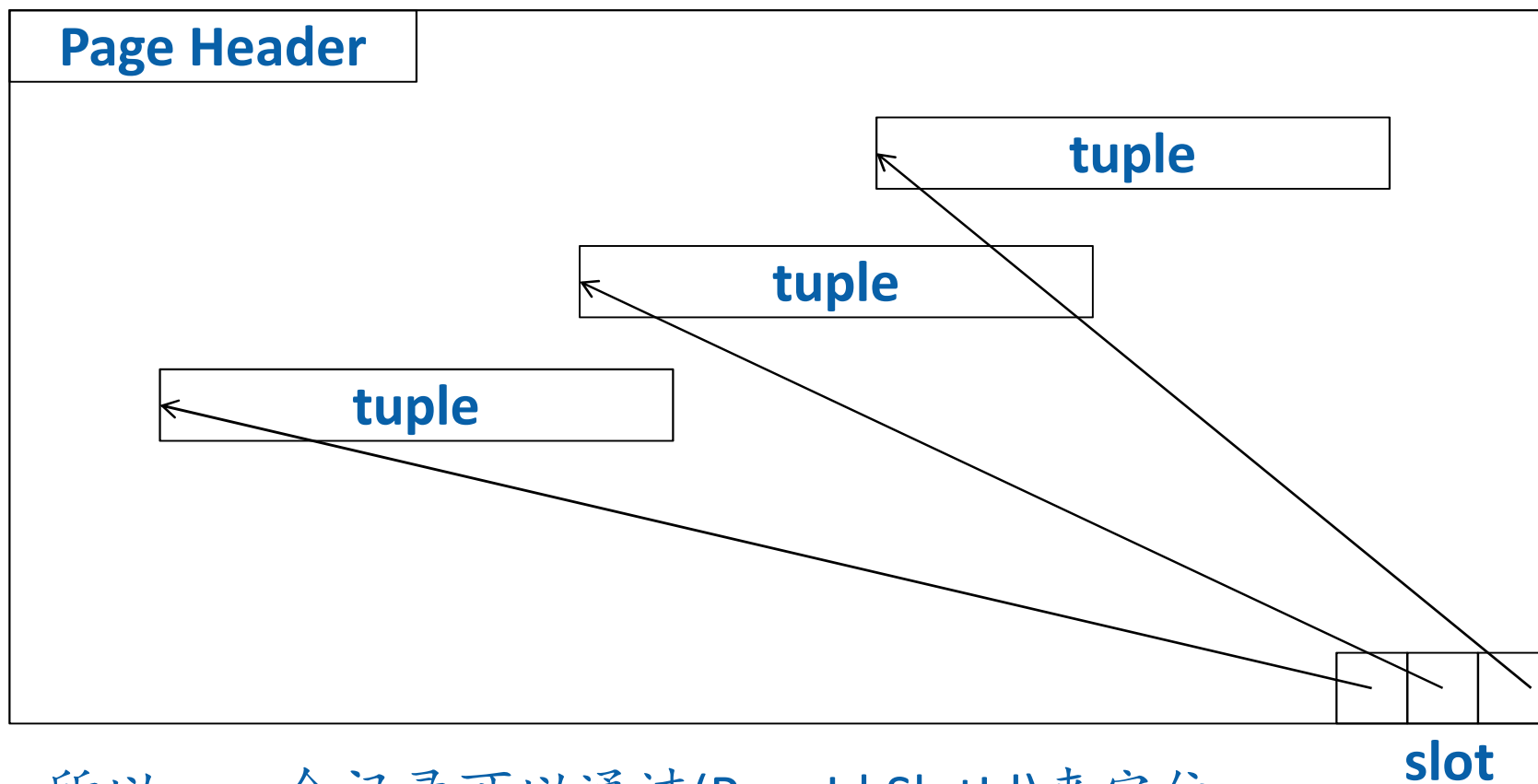
- 数据库系统内部架构概述
- 数据存储与访问路径概述
- 硬件和操作系统支持
- 磁盘空间管理
- 记录文件格式
  - 行式文件页结构
  - 行式记录结构
  - 列式文件结构
  - 顺序读和I/O模型
- 缓冲区管理

# Page内部结构 (Slotted Page)



- 方便存储变长的记录
- 记录超出页面大小就需要特殊处理
- Page Header包含: PageID, 校验和, LSN(事务日志序号)等

# Page内部结构 (Slotted Page)



- 所以，一个记录可以通过(PageId,SlotId)来定位
- 在一些系统中，RecordId就是这两部分的结合

# 行式记录结构

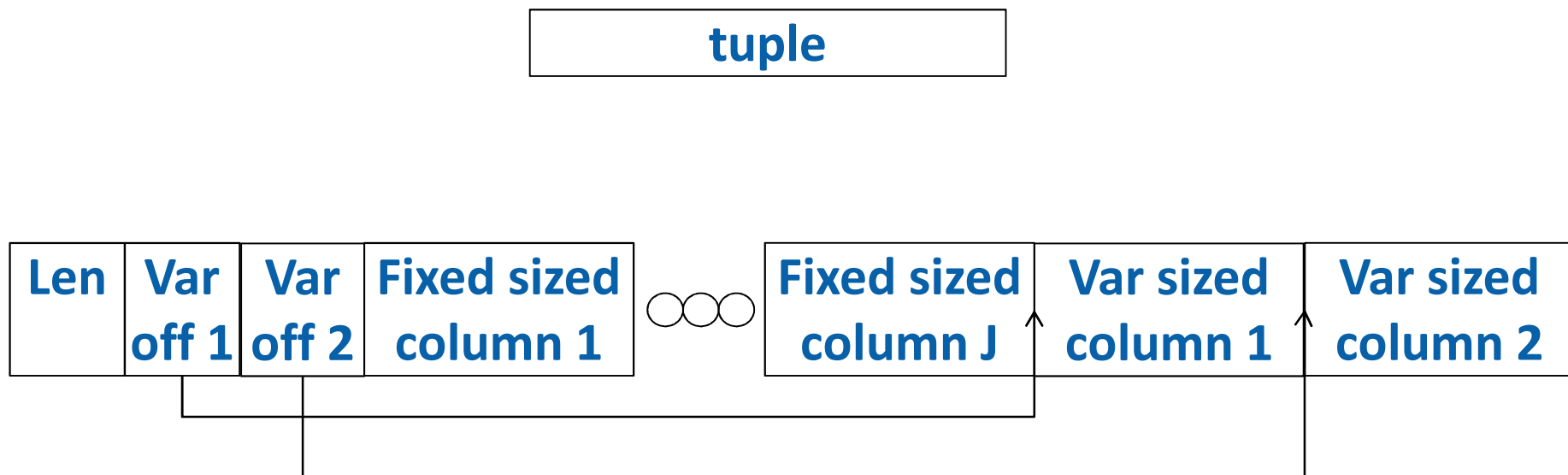
- 定长的列

- 数值类型：整数、浮点数等
- 定长字符串：char(n)
- 时间日期等

- 变长的列

- 变长字符串：varchar(n)
- 变长二进制对象：BLOB (Binary Large Object) 等

# Tuple的结构



- 举例：有两个变长的列



# 举例

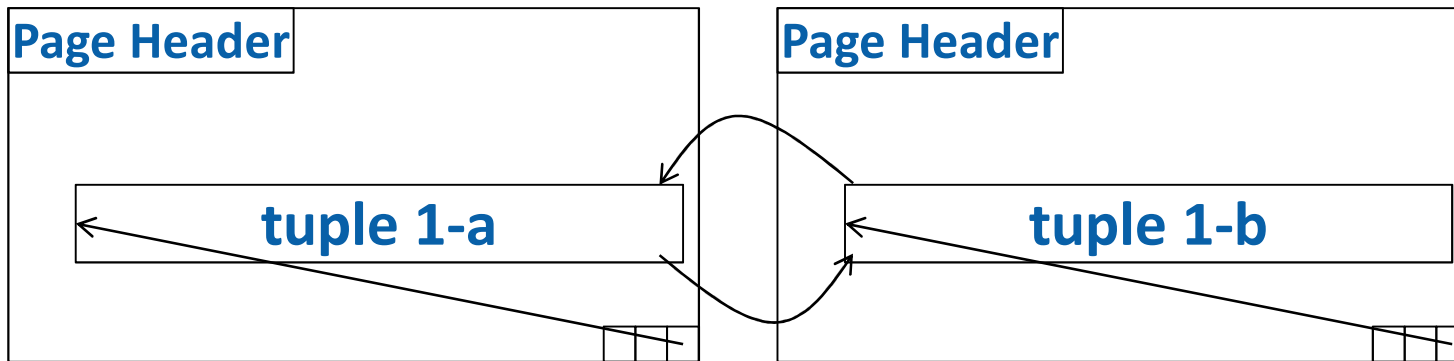
ID	Name	Birthday	Gender	Major	Year	GPA
131234	张飞	1995/1/1	男	计算机	2013	85

create table *Student* (  
    *ID* integer NOT NULL, *Name* varchar(20), *Birthday* date,  
    *Gender* enum(M, F), *Major* varchar(20), *Year* year, *GPA* float,  
    primary key (*ID*));

0	2	4	6	10	14	15	19	23	27	33
33	23	27	131234	1995/1/1	男	2013	85	张飞	计算机	
2B	2B	2B	4B	4B	1B	4B	4B	4B	6B	

# 跨块记录

- 当记录比一个数据页大时
- 分为多个子记录，存储在不同的数据页中
- 多个子记录之间互相指向



# Insertion

- 无序的堆文件

- 找到一个具有足够空间的Page
- 插入新记录

- 排序文件

- 找到该记录应该放置的Page
- 如果此Page中有足够的空间，那么直接插入新记录
  - 可以移动Page中现有的记录，来获得完整的空闲空间
- 如果空间不够
  - 在邻近Page中找空间，移动记录
  - 把当前Page分裂为两个Page，Page header包含溢出块的地址

# Deletion

- 找到对应的Page
- 删除Page中的记录
- 有时需要保留对应的slot，记录一个删除标记
  - 从而保证这个RecordId不被使用

# Update

- 如果需要的修改不会使原记录变大
  - 那么就在原记录位置修改
- 如果变大，那么就遇到与插入相似的问题
  - 当前Page有足够空间时，可以通过移动Page内部的Tuple来得到一片连续的空间放置记录
  - 当前Page没有足够的空间时，就需要考虑邻近Page或溢出

# 数据的顺序访问

```
select Name, GPA  
from Student  
where Major = '计算机';
```

- 顺序读取Student表的每个page
- 对于每个page，顺序访问每个tuple
- 检查条件是否成立
- 对于成立的读取Name和GPA

# 计算I/O模型

- 当一个操作中I/O的开销为主导地位时
- 考虑磁盘I/O次数（访问Page的个数）来近似算法时间

# 数据的顺序访问的I/O代价

```
select Name, GPA  
from Student  
where Major = '计算机';
```

- 假设Student表有M个Page
- 那么前述顺序访问的代价就为M



# 列式数据存储

- 每个列产生一个文件，存储所有记录中该列的值

ID	Name	Birthday	Gender	Major	Year	GPA
131234	张飞	1995/1/1	男	计算机	2013	85
145678	貂蝉	1996/3/3	女	经管	2014	90
129012	孙权	1994/5/5	男	法律	2012	80
121101	关羽	1994/6/6	男	计算机	2012	90
142233	赵云	1996/7/7	男	计算机	2014	95

存储为7个列文件

ID	Name	Birthday	Gender	Major	Year	GPA
131234	张飞	1995/1/1	男	计算机	2013	85
145678	貂蝉	1996/3/3	女	经管	2014	90
129012	孙权	1994/5/5	男	法律	2012	80
121101	关羽	1994/6/6	男	计算机	2012	90
142233	赵云	1996/7/7	男	计算机	2014	95

# 为什么要用列式存储？

- 数据库的分析查询

- 大部分情况只涉及一个表的少数几列
- 会读一大部分记录

- 在这种情况下，行式存储需要读很多无用的数据

- 采用列式存储可以降低读的数据量

ID	Name	Birthday	Gender	Major	Year	GPA
131234	张飞	1995/1/1	男	计算机	2013	85
145678	貂蝉	1996/3/3	女	经管	2014	90
129012	孙权	1994/5/5	男	法律	2012	80
121101	关羽	1994/6/6	男	计算机	2012	90
142233	赵云	1996/7/7	男	计算机	2014	95

# 列式存储的压缩

- 每个文件存储相同数据类型的值
- 数据更容易被压缩
- 比行式存储有更高的压缩比

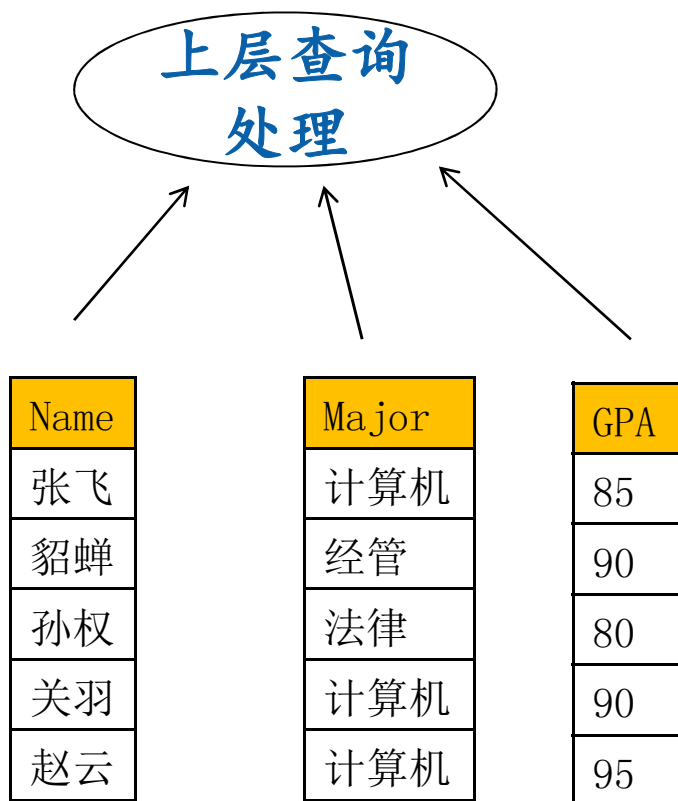
Year
2013
2014
2012
2012
2014

可以有多种简单的方法压缩

# 顺序访问节省I/O

```
select Name, GPA
from Student
where Major = '计算机';
```

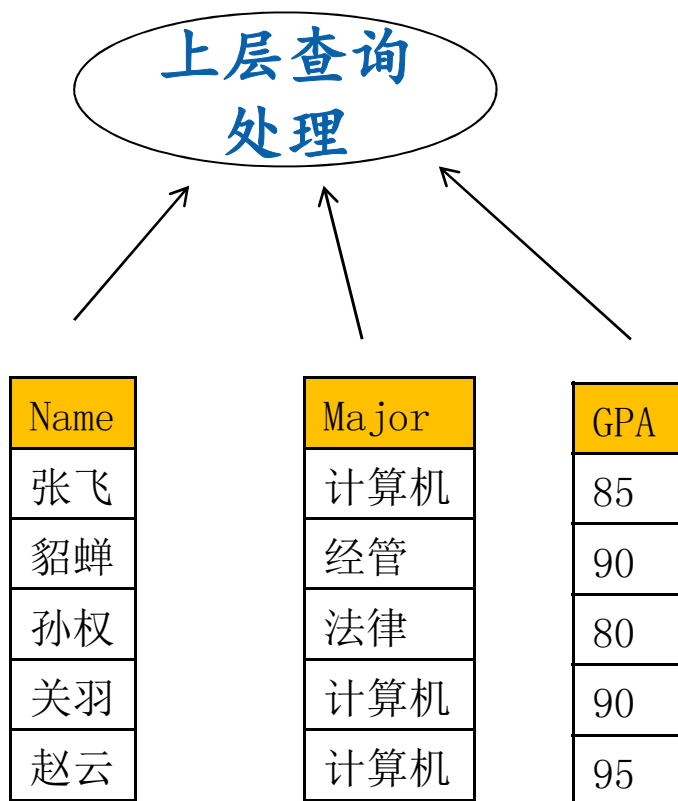
- 如果用到了一个表的多个列
- 多列需要拼装在一起，付出拼装代价



# 需要Join多列

```
select Name, GPA
from Student
where Major = '计算机';
```

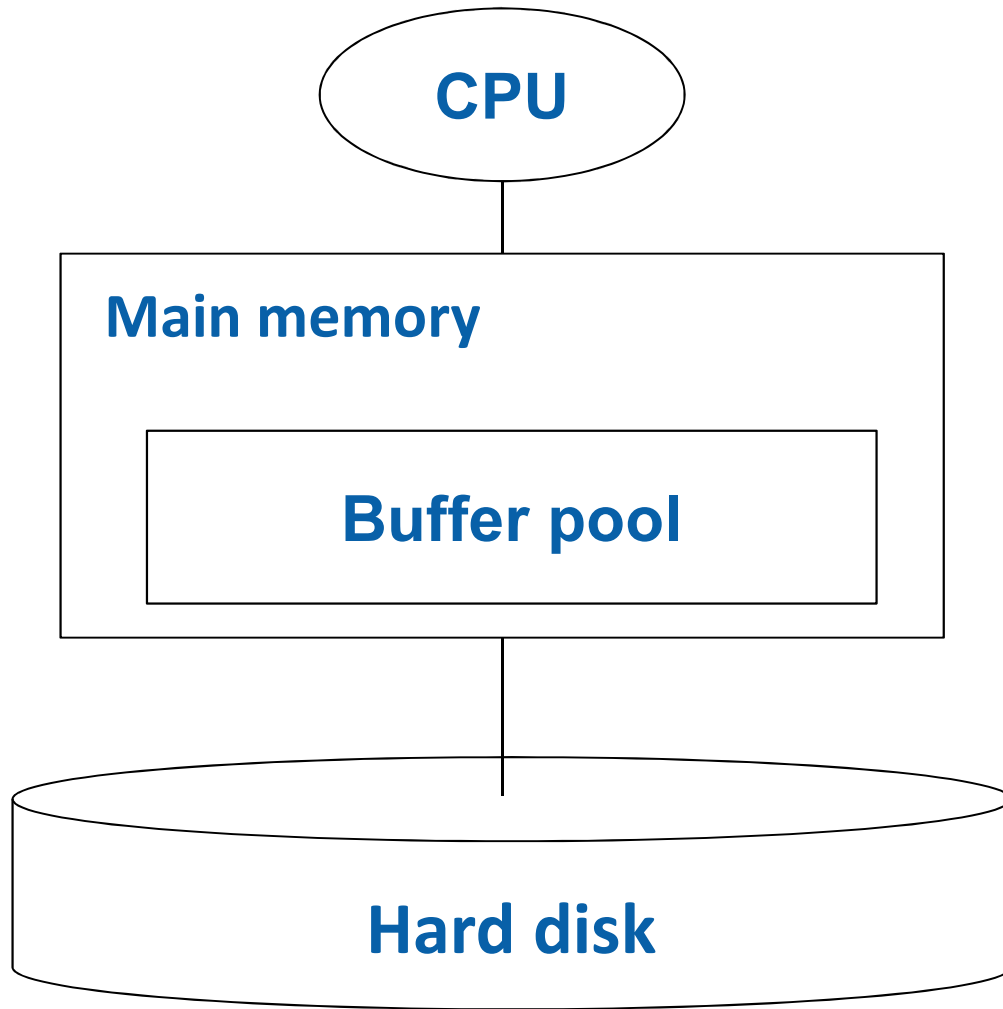
- 如果用到了一个表的多个列
- 多列需要拼装在一起，付出拼装代价



# Outline

- 数据库系统内部架构概述
- 数据存储与访问路径概述
- 硬件和操作系统支持
- 磁盘空间管理
- 记录文件格式
- 缓冲区管理

# 什么是Buffer Pool?



为什么需要Buffer pool?  
每次访问直接读写硬盘  
会有什么问题吗?

提高性能, 减少I/O

# 数据访问的局部性(locality)

- Temporal locality (时间局部性)

- 同一个数据元素可能会在一段时间内多次被访问

- ☞ Buffer pool

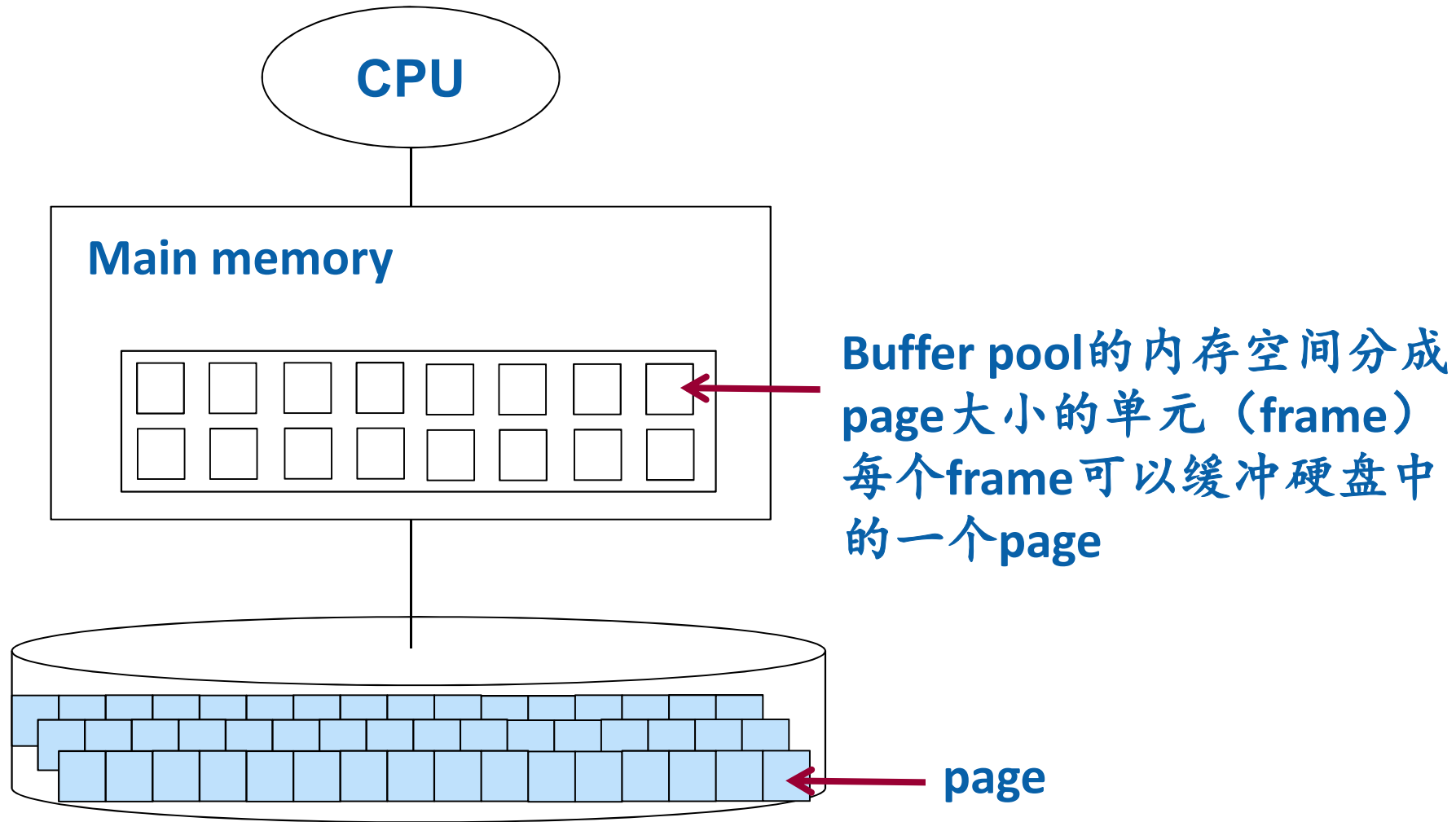
- Spatial locality (空间局部性)

- 位置相近的数据元素可能会被一起访问

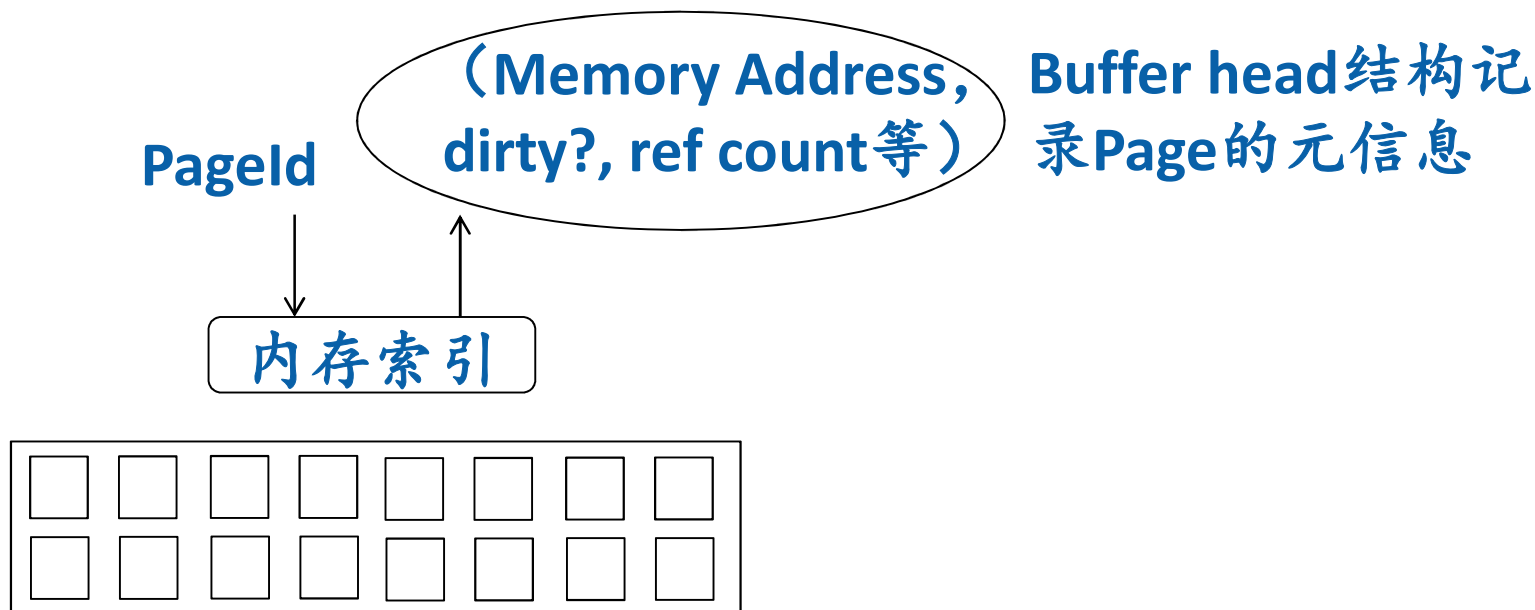
- ☞ Page为单位读写



# Buffer Pool的组成



# Buffer Pool的内存结构



- 每次访问一个Page都需要查找PageId索引
- 如果存在，那么返回内存地址，进行访问
  - 增计ref count，根据是否是写操作请求，记录dirty状态

# 访问Page A的过程

- 检查Page A是否在buffer pool之中
- 是：buffer pool hit
  - 直接访问buffer pool中的page A
    - 节省了I/O操作
- 否：buffer pool miss
  - 在buffer pool中找到一个可用的frame
  - 从硬盘读page A，放入这个frame



# Page Replacement (替换)

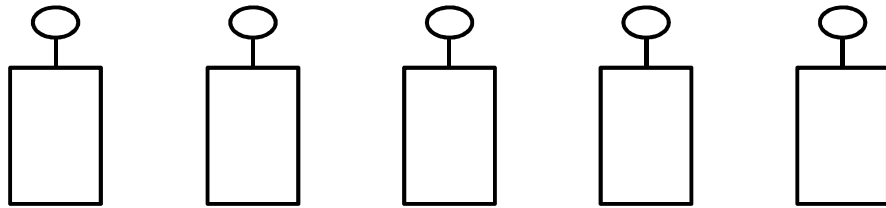
- 如果没有空闲的frame, 那么怎么办?
- 需要找一个已缓存的page, 替换掉
  - 这个page被称作Victim page
  - 如果这个page 被修改过 (dirty), 那么需要写回硬盘
- 替换策略? (如何选择Victim?)
  - 目标: 尽量减少I/O代价, 希望Victim在近期不可能被访问
  - 算法: 通常是LRU (Least Recently Used) 的某种变形

# Replacement Policies(替换策略)

- 操作系统课应该讲，常见的替换策略有
  - Random：随机替换
  - FIFO(First In First Out)：替换最老的页
  - LRU (Least Recently Used)：最近最少使用
- 我们围绕LRU介绍数据库中常见的算法

# LRU (Least Recently Used)

- LRU的实现方法1

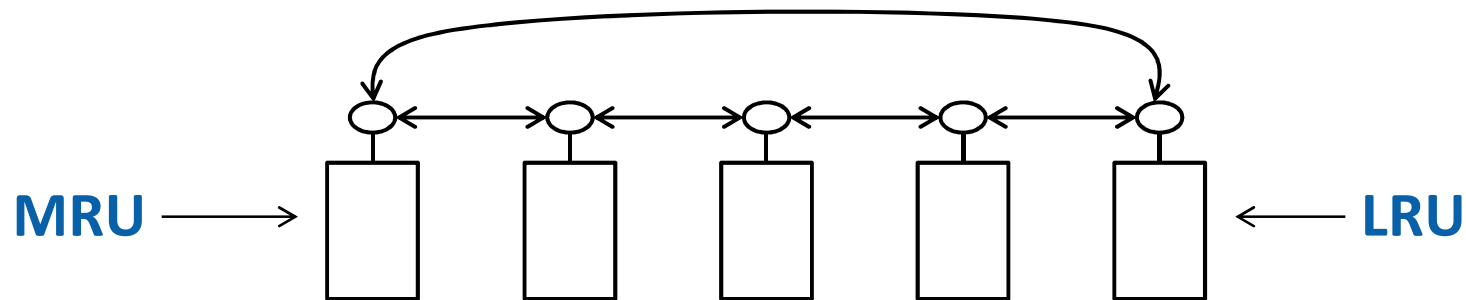


- Buffer head记录访问时间戳
- 替换：找到时间戳最早（小）的页为Victim
- 问题：替换操作是 $O(N)$ !



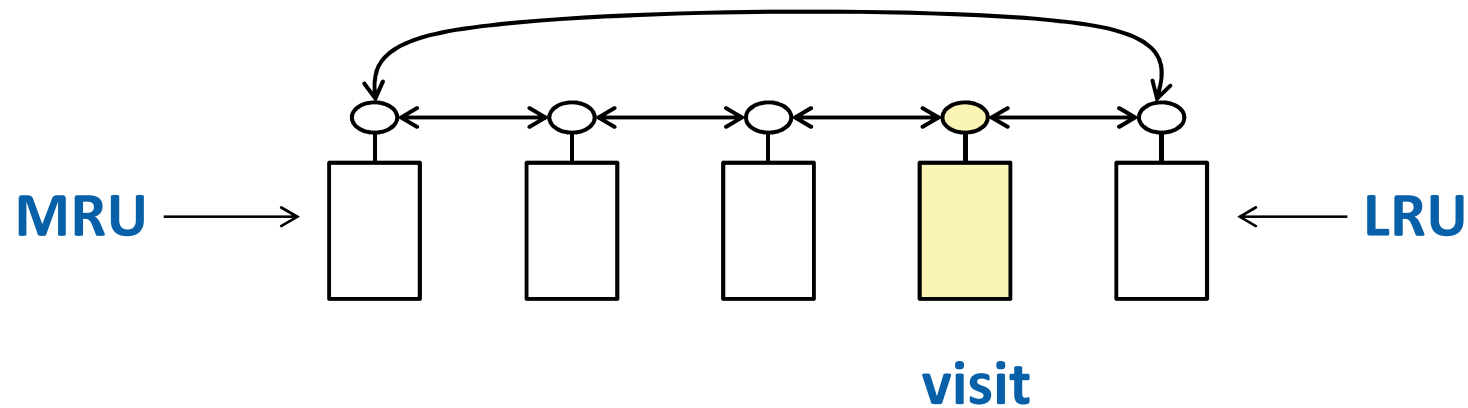
# LRU (Least Recently Used)

- LRU的实现2



# LRU (Least Recently Used)

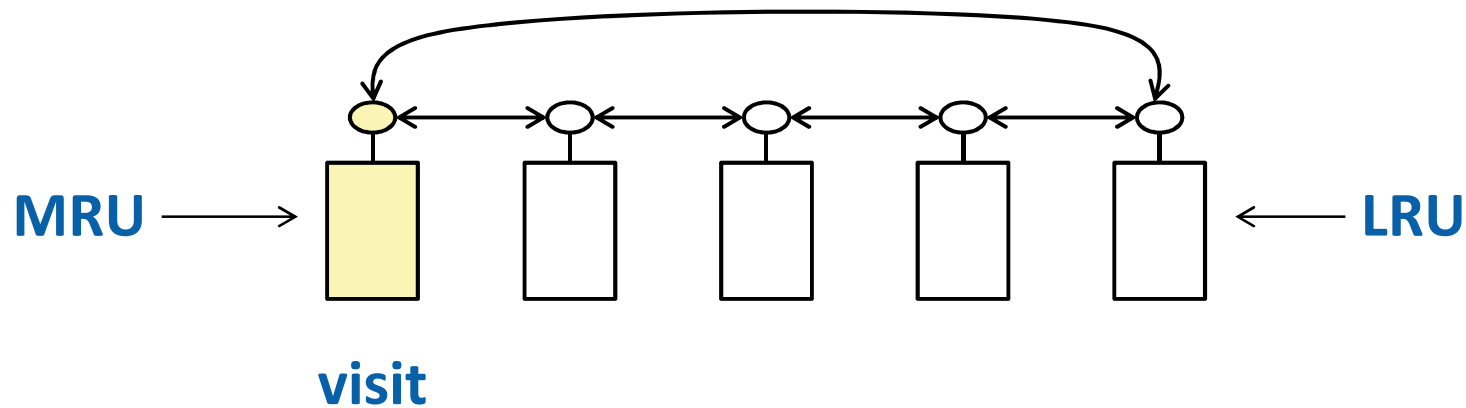
- LRU的实现2





# LRU (Least Recently Used)

- LRU的实现2

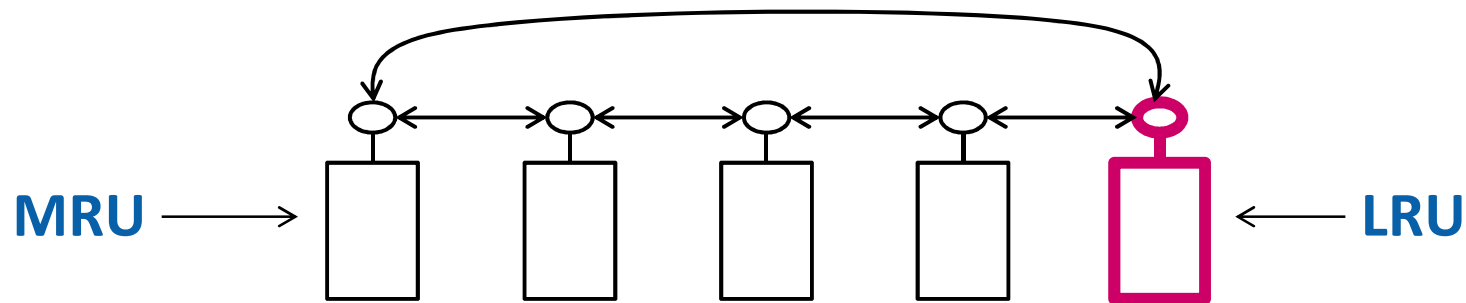


当一页被访问时，把它移动到最前端



# LRU (Least Recently Used)

- LRU的实现2

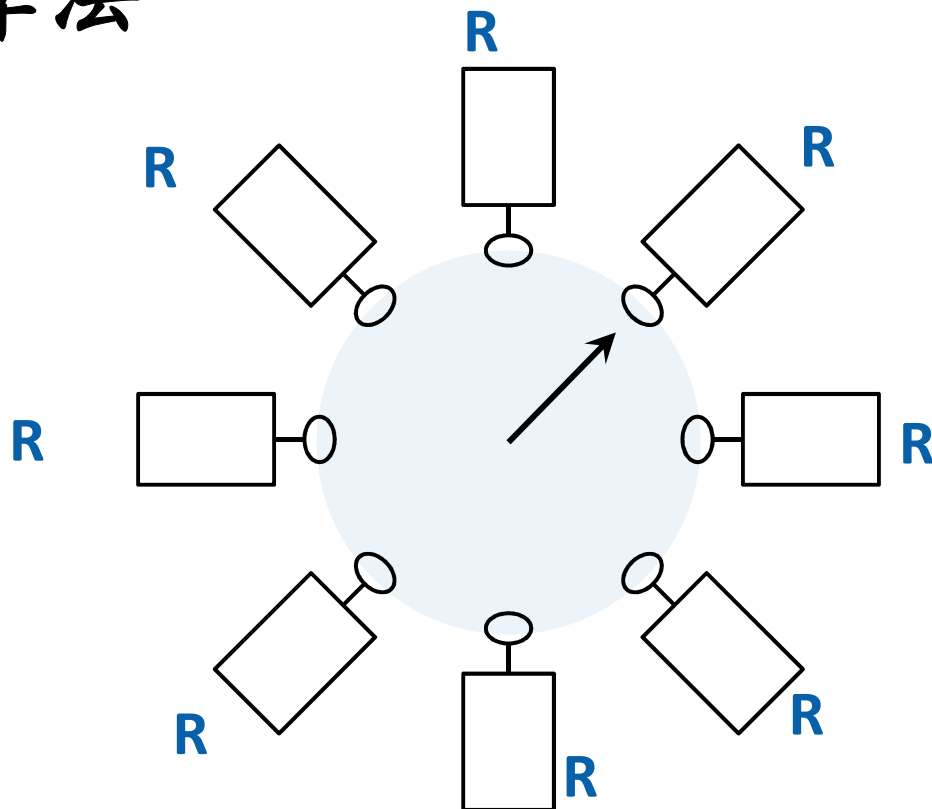


替换：总是选择最后一个Page为Victim  
 $O(1)$ 代价😊

但是：修改队列的代价，多线程共享队头

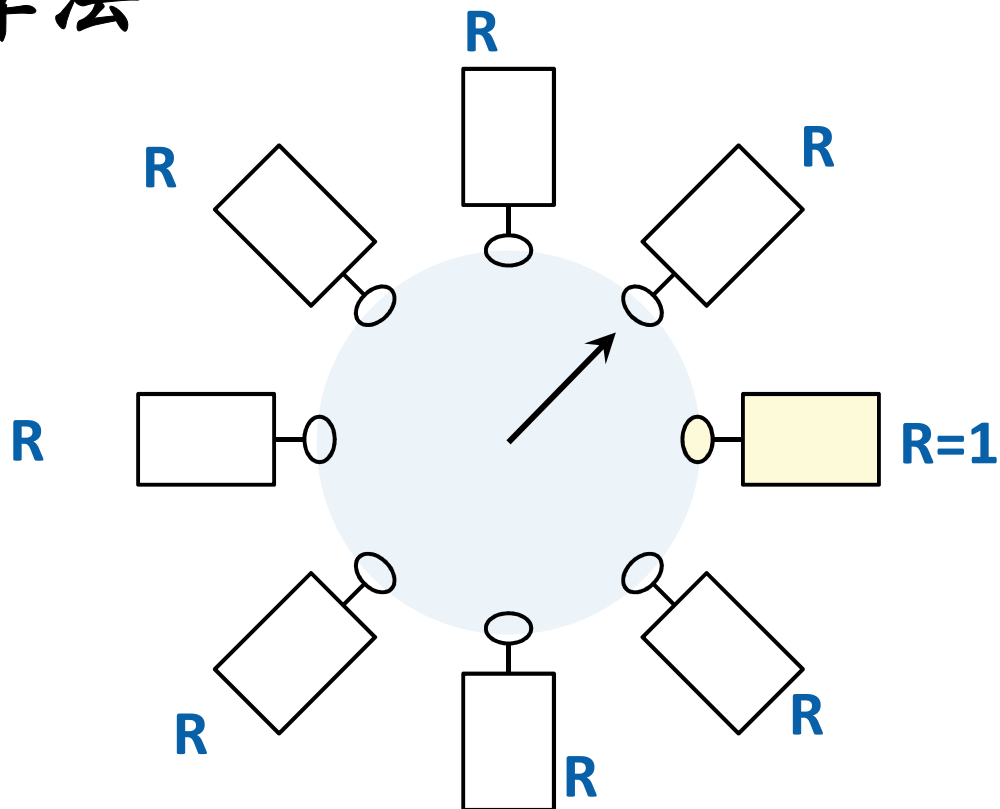


# Clock算法



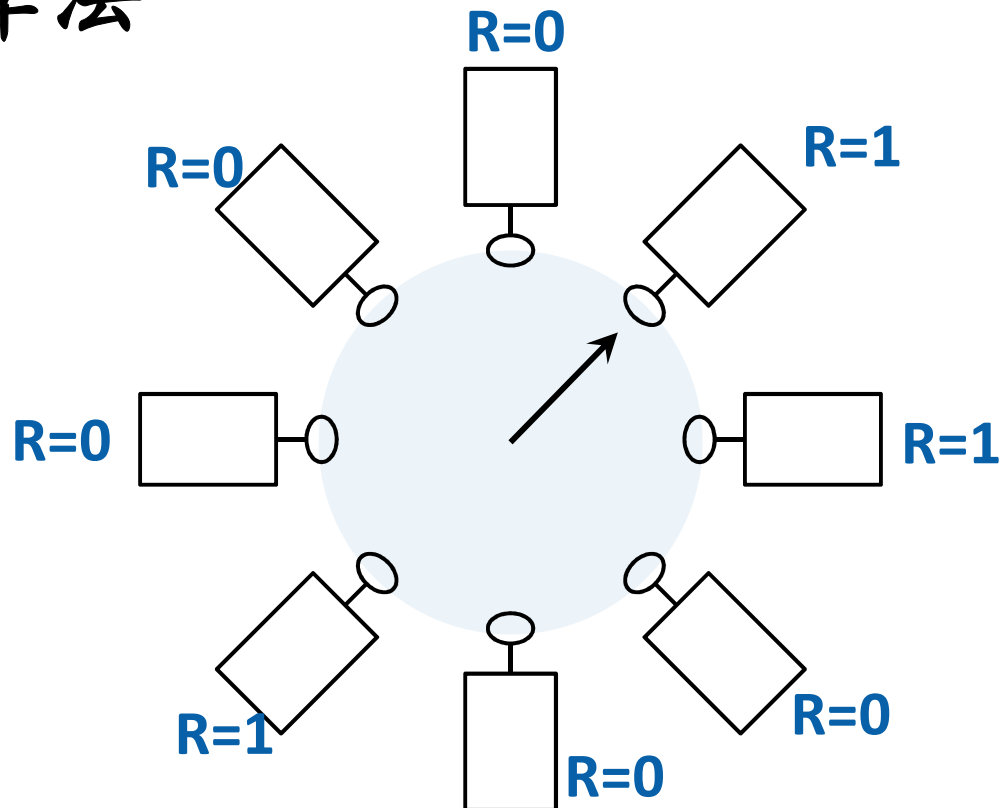
- 数据结构：Buffer head记录R，取值为0或1

# Clock算法



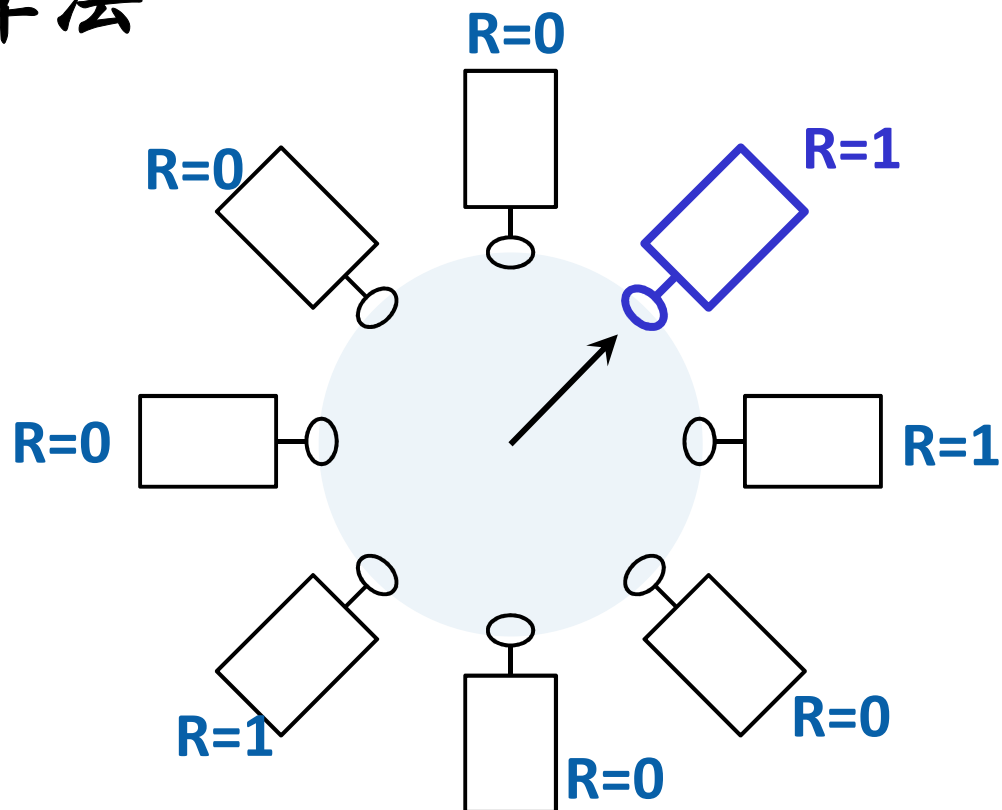
- 访问一个页：赋值 $R=1$

# Clock算法



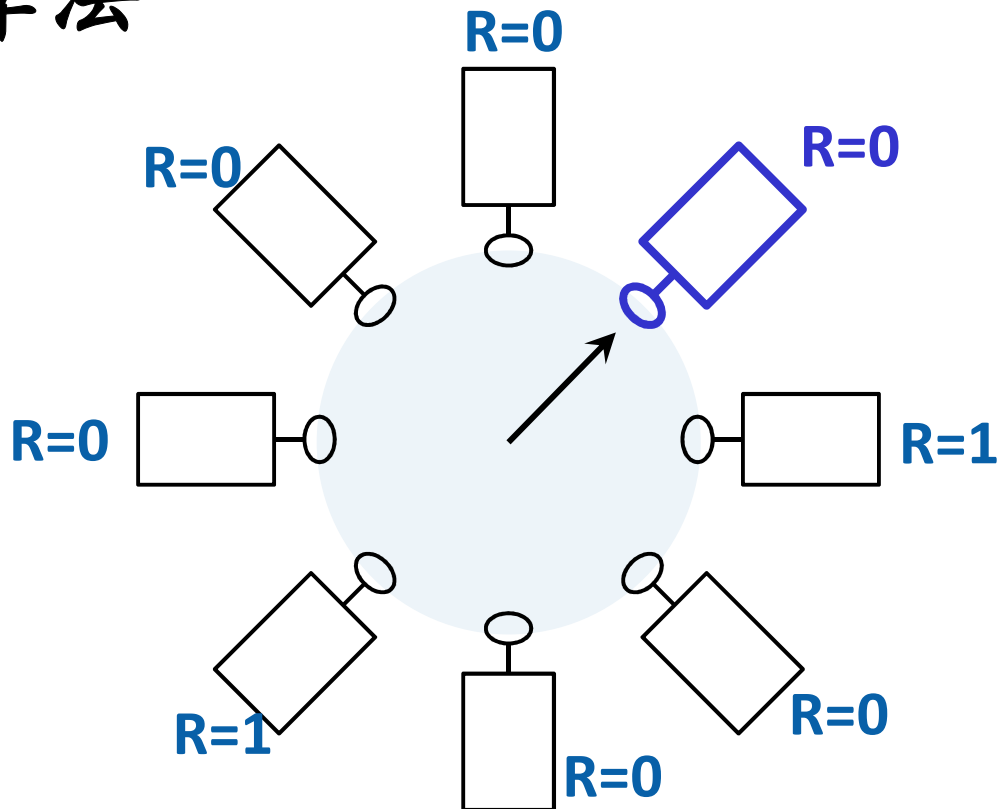
- 替换，顺时针旋转，依次查看下一个页
- if ( $R == 1$ ) then { $R=0$ ; 继续旋转;}

# Clock算法



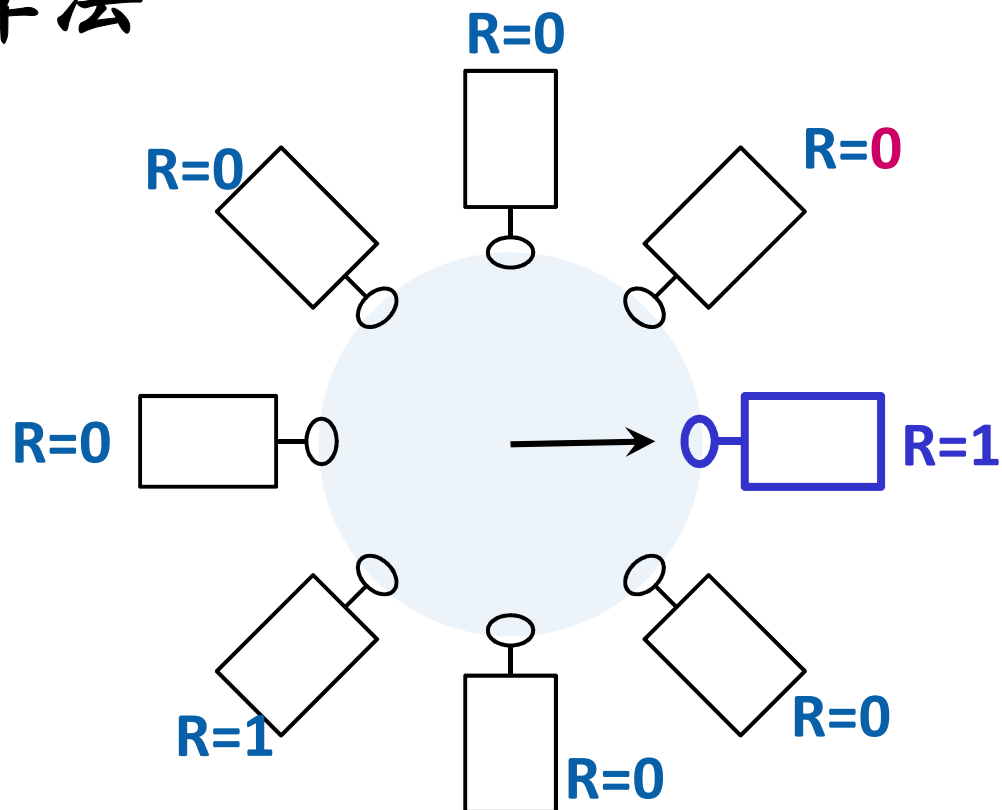
- 替换，顺时针旋转，依次查看下一个页
- if ( $R == 1$ ) then { $R=0$ ; 继续旋转;}

# Clock算法



- 替换，顺时针旋转，依次查看下一个页
- if ( $R == 1$ ) then { $R=0$ ; 继续旋转;}

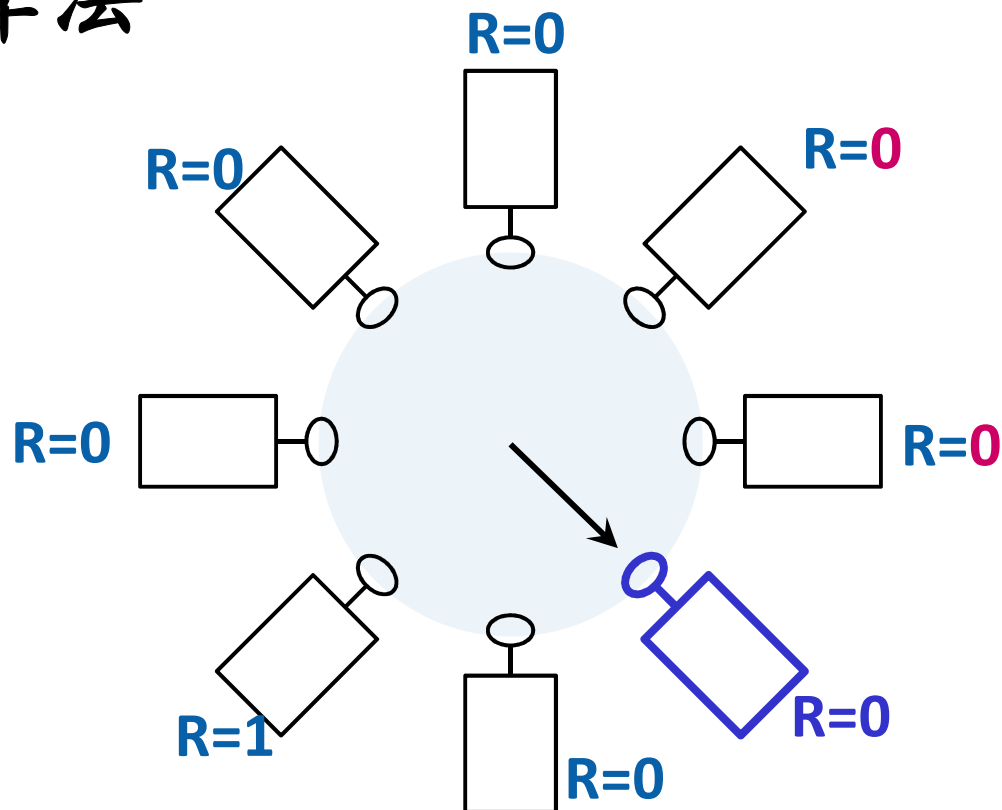
# Clock算法



- 替换，顺时针旋转，依次查看下一个页
- if ( $R == 1$ ) then { $R=0$ ; 继续旋转;}

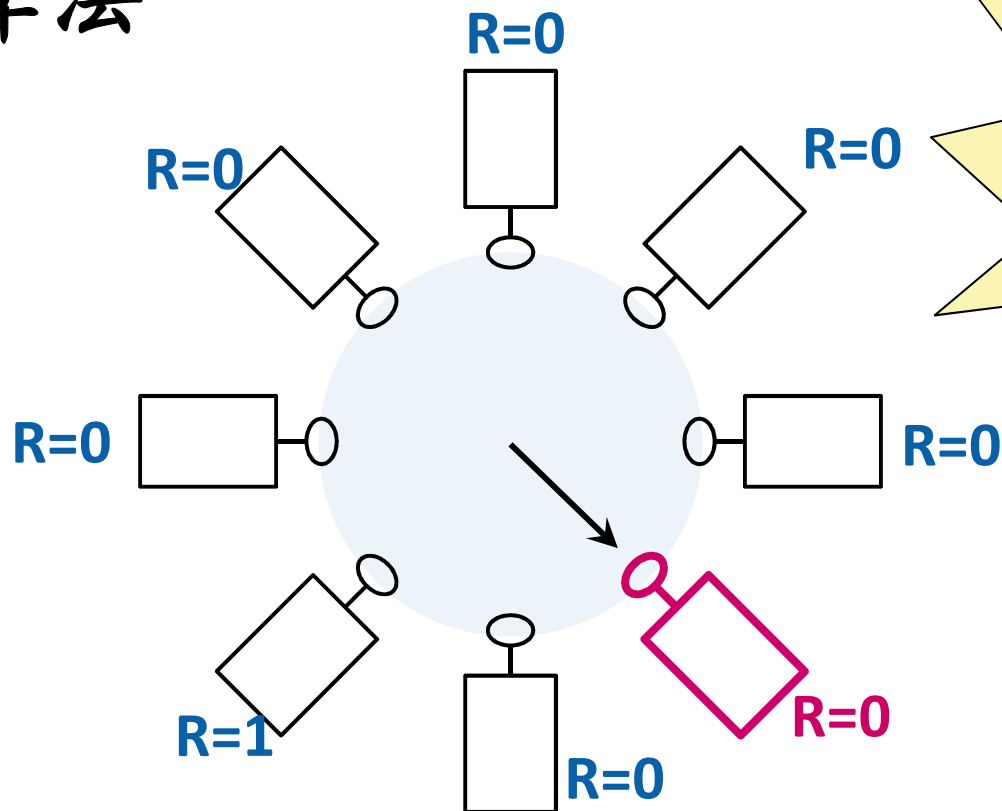


# Clock算法



- 替换，顺时针旋转，依次查看下一个页
- if ( $R == 1$ ) then { $R=0$ ; 继续旋转;}
- if ( $R == 0$ ) then 选中为Victim

# Clock算法



- 替换, 顺时针旋转, 依次查看下一个页
- if ( $R == 1$ ) then { $R=0$ ; 继续旋转;}
- if ( $R == 0$ ) then 选中为Victim

# 小结

- 数据库系统内部架构概述
- 数据存储与访问路径概述
  - 存储层次
  - 存储介质：磁盘、固态硬盘等
  - 磁盘阵列
  - 操作系统支持
  - 数据与索引
- 磁盘空间管理：工作原理
- 记录文件格式
  - 行式文件页结构
  - 行式记录结构
  - 列式文件结构
  - 顺序读和I/O模型
- 缓冲区管理：工作原理，替换算法