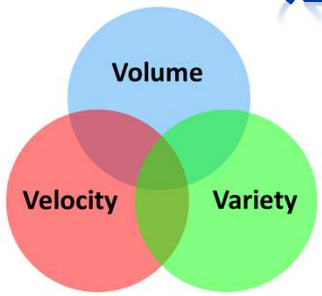
大数据系统与大规模数据分析

关系型数据管理系统（2）



陈世敏

中科院计算所
计算机体系结构
国家重点实验室

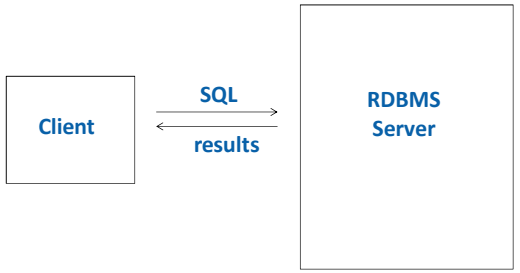
Outline

- 数据库系统架构
- 数据存储与访问
  - 数据表
  - 索引
  - 缓冲池
- 运算的实现
  - Operator tree
  - Selection & Projection
  - Join

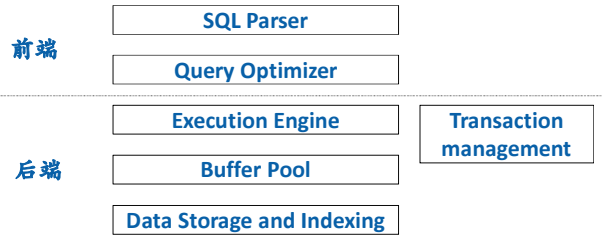
DBMS

- Database Management System
  (数据库管理系统)
- RDBMS: Relational Database Management System
  (关系型数据库系统)
- 目前的三大主流商用系统
  - Oracle, Microsoft SQL Server, IBM DB2
- 开源数据库系统
  - PostgreSQL, MySQL
  - Library: sqlite

通常的系统为典型的Client / Server

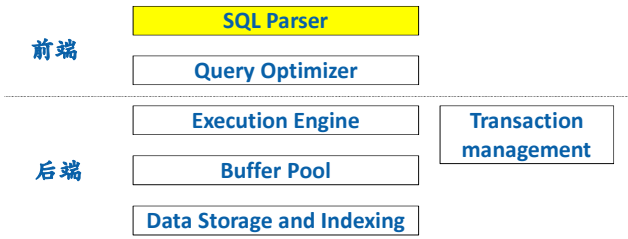


RDBMS的系统架构(单机)



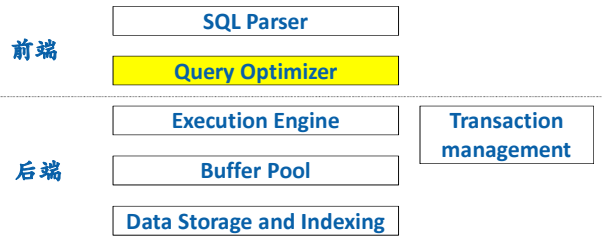
RDBMS的系统架构

- 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



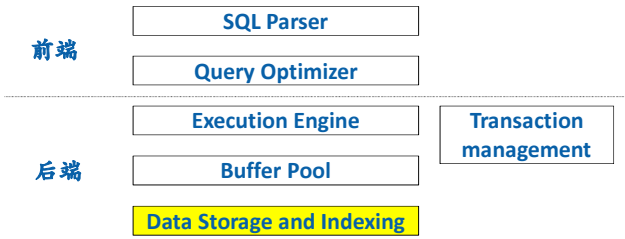
RDBMS的系统架构

- 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



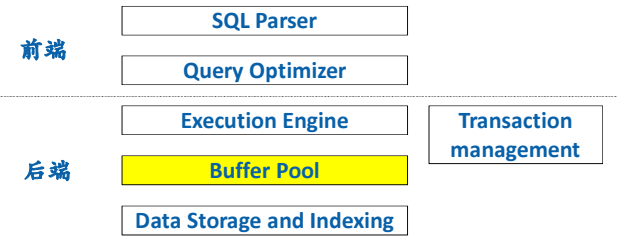
RDBMS的系统架构

- 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



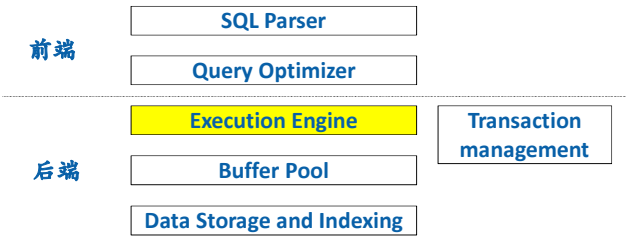
RDBMS的系统架构

- Buffer pool: 在内存中缓存硬盘的数据



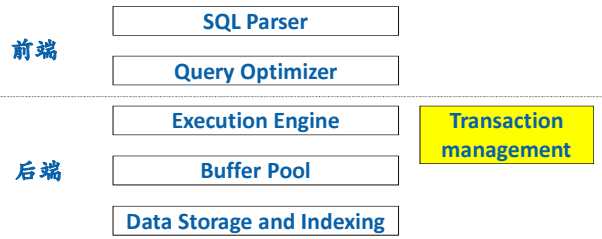
单机RDBMS的系统架构

- 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



RDBMS的系统架构

- Transaction management：事务管理
  - 目标是实现ACID
  - 进行logging写日志，locking加锁
  - 保证并行transactions事务的正确性



Outline

- 数据库系统架构
- 数据存储与访问
  - 数据表(Table)
  - 索引(Index)
  - 缓冲池(Buffer pool)
- 运算的实现
  - Operator tree
  - Selection & Projection
  - Join

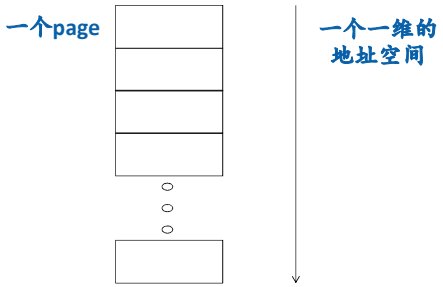
数据库 vs. 文件系统（数据存储角度比较）

- 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
- 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
- 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

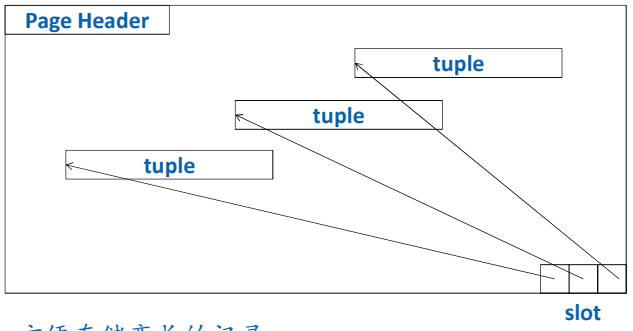
- 硬盘最小存储访问单位为一个扇区：512B
- 文件系统访问硬盘的单位通常为：4KB
- RDBMS最小的存储单位是database page size
  - Data page size 可以设置为1~多个文件系统的 page
  - 例如，4KB, 8KB, 16KB, …
  - 我们下面用page简称database page

数据在硬盘上的存储



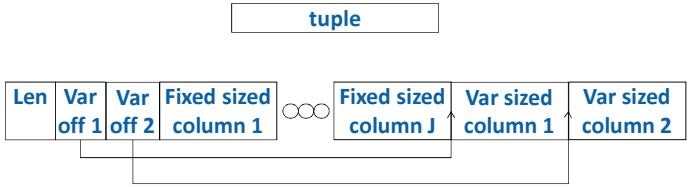
- Raw partition或file

Page内部结构



- 方便存储变长的记录
- 记录超出页面大小就需要特殊处理

Tuple的结构

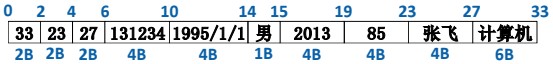


- 举例：有两个变长的列

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*));



数据的顺序访问

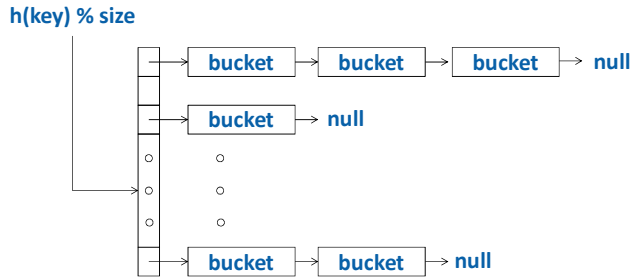
```
select Name, GPA
from Student
where Major = '计算机';
```

- 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
- 有什么性能问题吗？ **如果全校有100个系会怎么样？**

Selective Data Access (有选择性的访问)

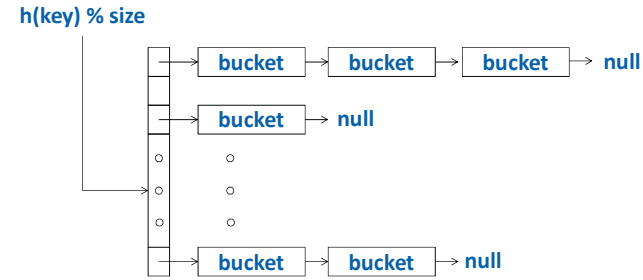
- 使用index(索引)
  - Tree based index
  - Hash based index
- 有什么不同？
  - Tree based：有序，支持点查询和范围查询
  - Hash based：无序，只支持点查询

Chained Hash Table



- h(key) 对key进行位运算产生一个整数
- size是hash table的header数组的元素个数

Chained Hash Table on Disk



在硬盘上怎么存？

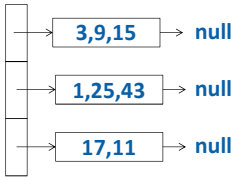
bucket = page

当chain上平均bucket数太多时，需要增大size，重新hashing

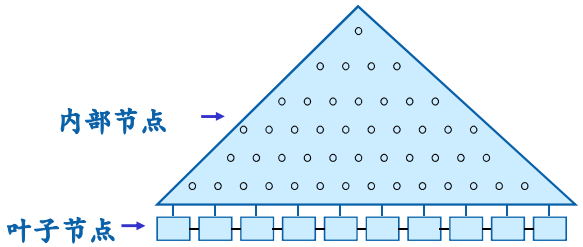
（存在hash table design可以降低re-hashing的代价）

举例

- Key: 1, 3, 25, 17, 9, 11, 43, 15
- h(key) = key
- Size = 3

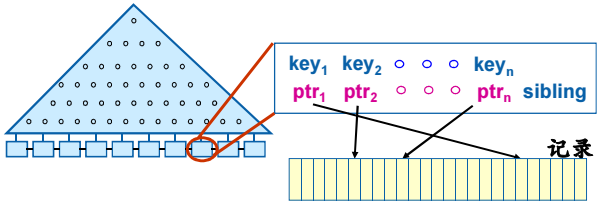


B+-Trees



- 每个节点是一个page
- 所有key存储在叶子节点
- 内部节点完全是索引作用

叶子节点

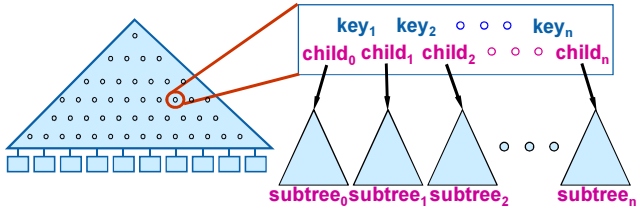


Keys 按照从小到大顺序排列: $key_1 < key_2 < \cdots < key_n$

叶节点自左向右也是从小到大排序，以sibling pointer链起来

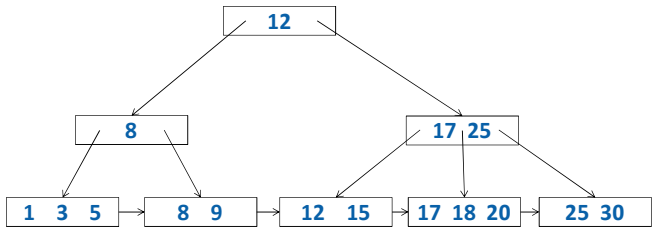
(ptr= record ID; sibling = page ID)

内部节点



$subtree_0 < key_1 \leq subtree_1 < key_2 \cdots < key_n$

举例

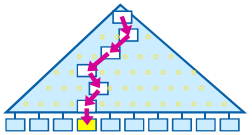


假设每个节点的child/pointer个数为B=3

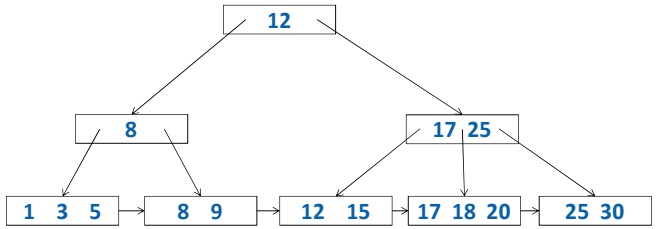
B+-Tree: Search

Search:

- 从根节点到叶节点
- 每个节点中进行二分查找
  - 内部节点：找到包括search key的子树
  - 叶节点：找到匹配

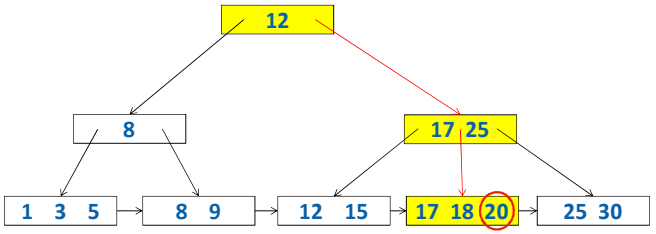


举例 Search(20)



假设每个节点的child/pointer个数为B=3

举例 Search(20)

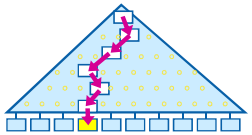


假设每个节点的child/pointer个数为B=3

B+-Tree: Search

Search代价

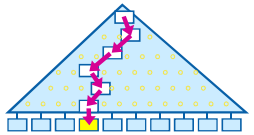
- 共有N个key
- 每个节点的child/pointer个数为B
- 总I/O次数=树高：$O(\log_B N)$
- 总比较次数
  - 每个节点内部二分查找：$O(\log_2 B)$
  - $O(\log_B N) \times O(\log_2 B) = O(\log_2 N)$



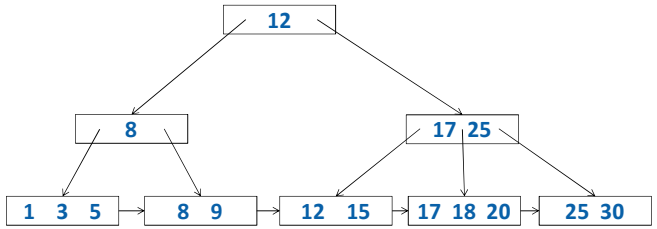
B+-Tree: Insertion

Insertion

- Search 然后在节点中插入
- 叶节点未满，插入叶节点
- 叶节点满了，node split(节点分裂)



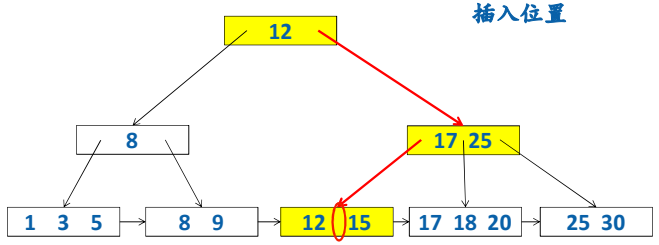
举例 Insert(14)



假设每个节点的child/pointer个数为B=3

举例 Insert(14)

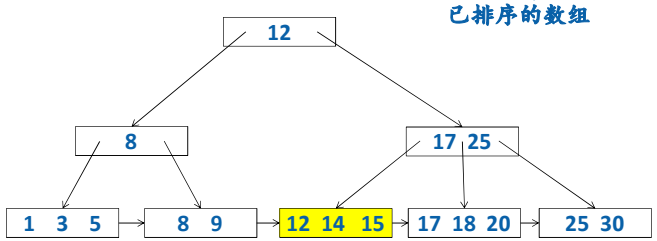
通过search找到插入位置



假设每个节点的child/pointer个数为B=3

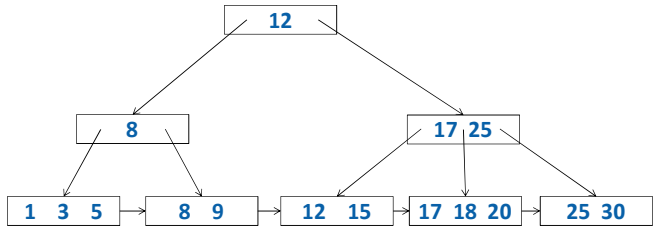
举例 Insert(14)

叶子未满，插入已排序的数组



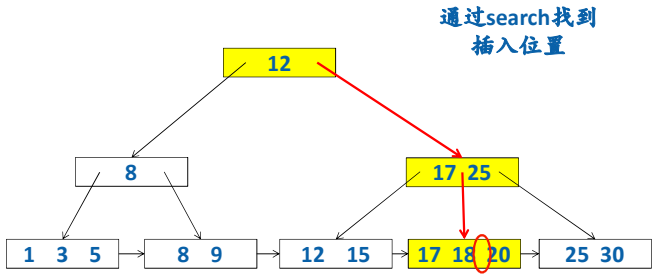
假设每个节点的child/pointer个数为B=3

举例 Insert(19)



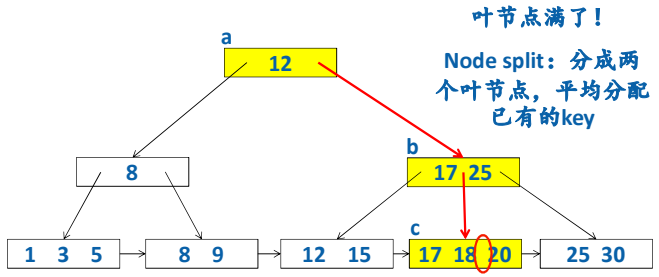
假设每个节点的child/pointer个数为B=3

举例 Insert(19)



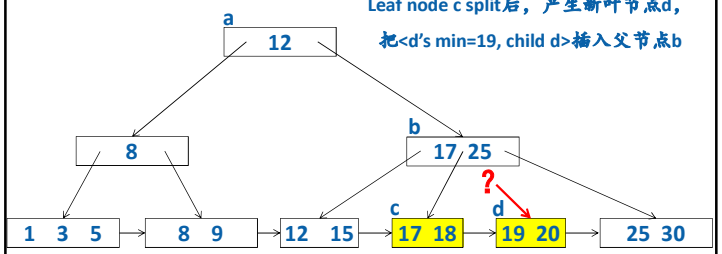
假设每个节点的child/pointer个数为B=3

举例 Insert(19)



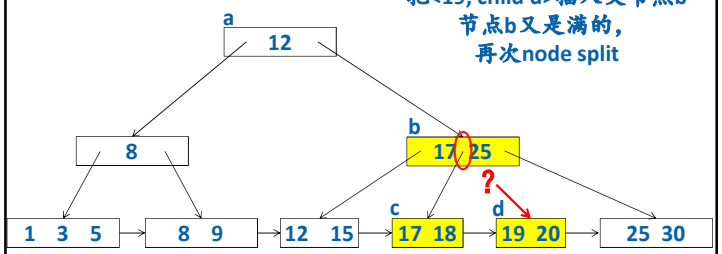
假设每个节点的child/pointer个数为B=3

举例 Insert(19)



假设每个节点的child/pointer个数为B=3

举例 Insert(19)

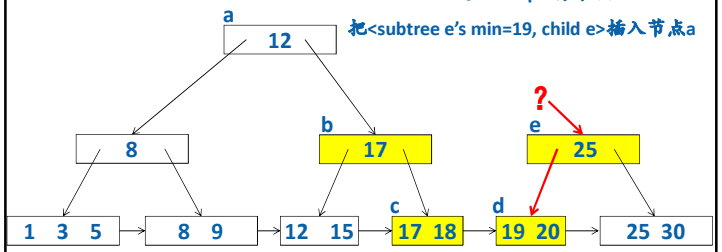


假设每个节点的child/pointer个数为B=3

举例 Insert(19)

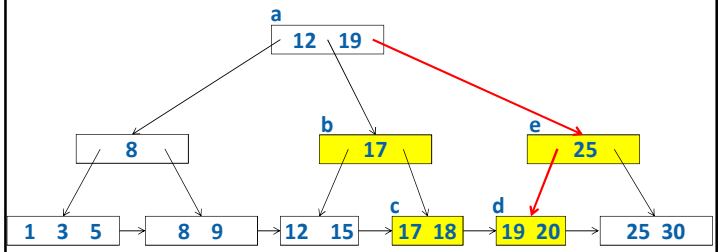
b 经过node split分裂出e

把<subtree e's min=19, child e>插入节点a



假设每个节点的child/pointer个数为B=3

举例 Insert(19)

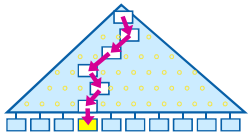


假设每个节点的child/pointer个数为B=3

B+-Tree: Deletion

Deletion

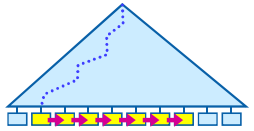
- Search 然后在节点中删除
- node merge?
  - 原设计：当节点中key个数小于一半
  - 实际实现：数据总趋势是增长的
    可以只有节点为空时才node merge
    或者完全不进行node merge



B+-Tree： Range Scan

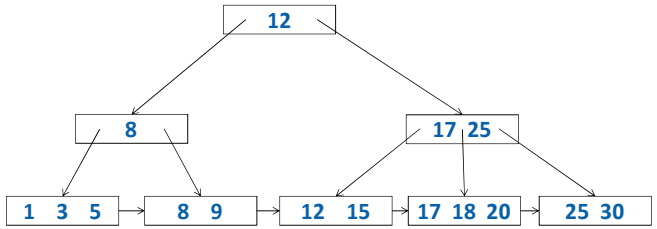
Range Scan

- 找到起始叶结点，包括范围起始值
- 沿着叶的链接读下一个叶结点
- 直至遇到范围终止值



举例

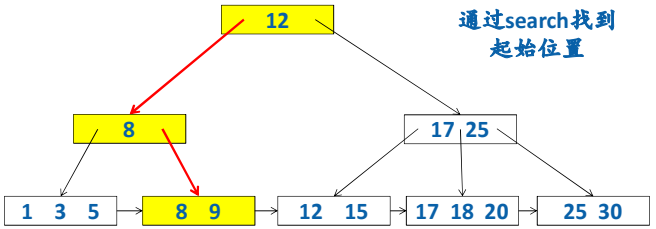
Range scan (9, 20):
获取[9, 20]区间的index entry



假设每个节点的child/pointer个数为B=3

举例

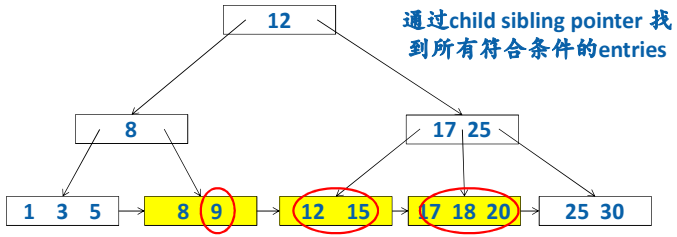
Range scan (9, 20):
获取[9, 20]区间的index entry



假设每个节点的child/pointer个数为B=3

举例

Range scan (9, 20):
获取[9, 20]区间的index entries



假设每个节点的child/pointer个数为B=3

Selective Data Access (有选择性的访问)

- 使用index(索引)
  - Tree based：有序，支持点查询和范围查询
  - Hash based：无序，只支持点查询
- Clustered index(主索引)与 Secondary index(二级索引)
  - Clustered: 记录就存在index中，记录顺序就是index顺序
  - Secondary: 记录顺序不是index顺序，index中存储page ID和in-page tuple slot ID.

索引数据访问

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

假设已经建立了以Major为key的二级索引

- 在二级索引中搜索*Major* = *'计算机'*
- 对于每个匹配项，访问相应的tuple
- 读取Name和GPA

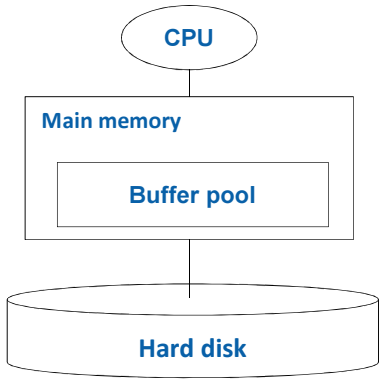
比较顺序访问与二级索引访问

- 顺序访问
  - 需要处理每一个记录
  - 顺序读每一个page
- 二级索引访问
  - 有选择地处理记录
  - 随机读相关的page

到底应该采用哪种方式呢？

- 由最终选中了多大比例的记录决定：selectivity
- 可以根据预测的selectivity、硬盘顺序读和随机读的性能，估算两种方式的执行时间
- 选择时间小的方案
- 这就是query optimizer的一个任务

什么是Buffer Pool?



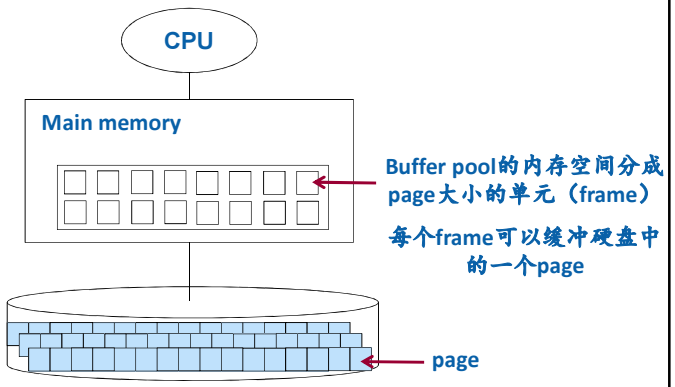
为什么需要Buffer pool?
每次访问直接读写硬盘会有什么问题吗？

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



访问一个Page A

- 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

Replacement (替换)

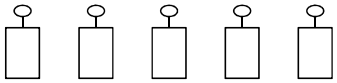
- 如果没有空闲的frame，那么怎么办？
- 需要找一个已缓存的page，替换掉
  - 这个page被称作Victim page
  - 如果这个page 被修改过，那么需要写回硬盘
- 替换策略？（如何选择Victim?）
  - 目标：尽量减少I/O代价，希望Victim在近期不可能被访问
  - 算法：通常是LRU（Least Recently Used）的某种变形

Replacement Policies(替换策略)

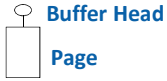
- 操作系统课应该讲，常见的替换策略有
  - Random：随机替换
  - FIFO(First In First Out)：替换最老的页
  - LRU (Least Recently Used)：最近最少使用
- 我们围绕LRU介绍数据库中常见的算法

LRU（Least Recently Used）

- LRU的实现方法1

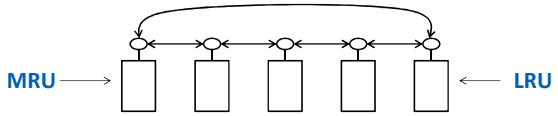


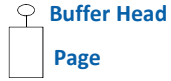
- Buffer head记录访问时间戳
- 替换：找到时间戳最早（小）的页为Victim
- 问题：替换操作是O(N)!



LRU（Least Recently Used）

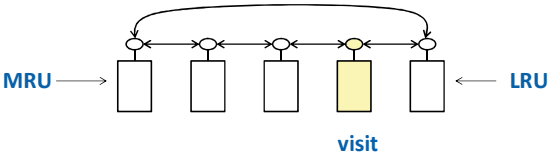
- LRU的实现2

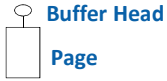




LRU（Least Recently Used）

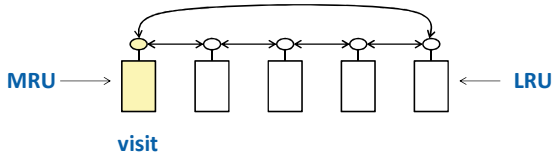
- LRU的实现2



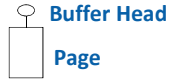


LRU（Least Recently Used）

- LRU的实现2

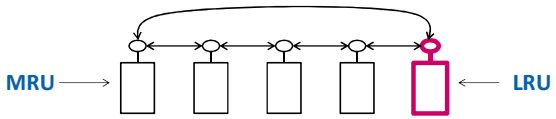


当一页被访问时，把它移动到最前端



LRU（Least Recently Used）

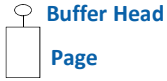
- LRU的实现2



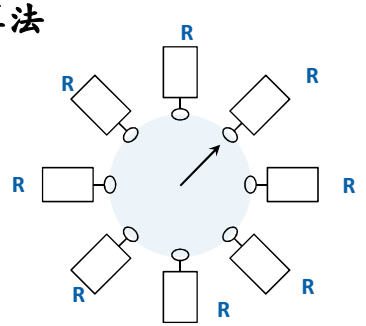
替换：总是选择最后一个Page为Victim

O(1)代价☺

但是：修改队列的代价，多线程共享队头

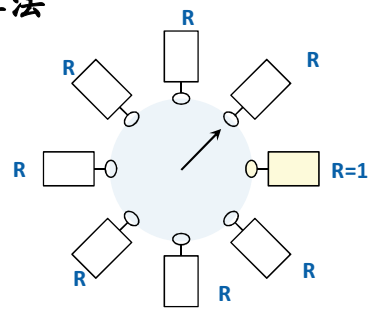


Clock算法



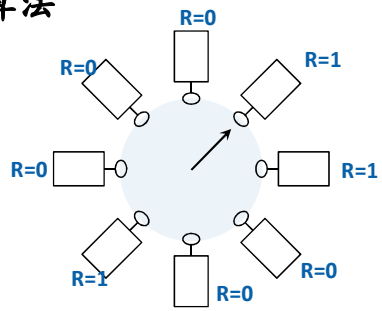
- 数据结构：Buffer head记录R，取值为0或1

Clock算法



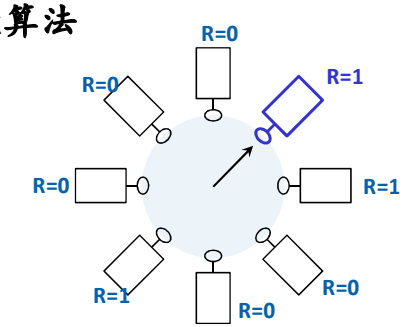
- 访问一个页：赋值R=1

Clock算法



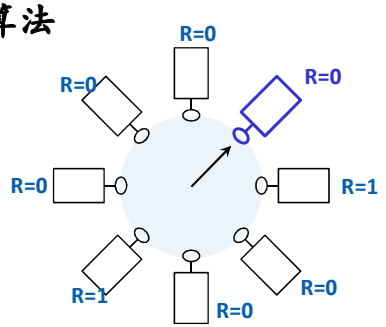
- 替换，顺时针旋转，依次查看下一个页
- if (R == 1) then {R=0; 继续旋转;}

Clock算法



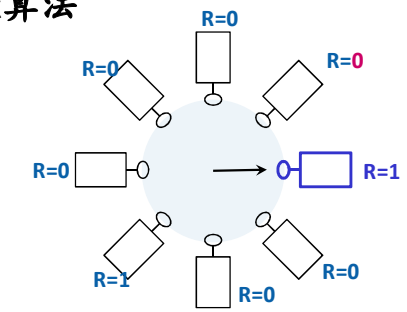
- 替换，顺时针旋转，依次查看下一个页
- if (R == 1) then {R=0; 继续旋转;}

Clock算法



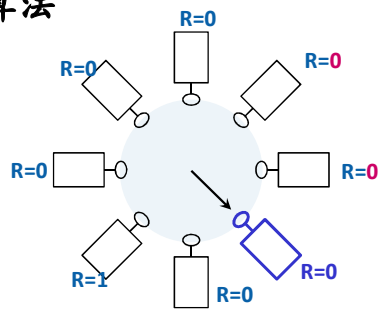
- 替换，顺时针旋转，依次查看下一个页
- if (R == 1) then {R=0; 继续旋转;}

Clock算法



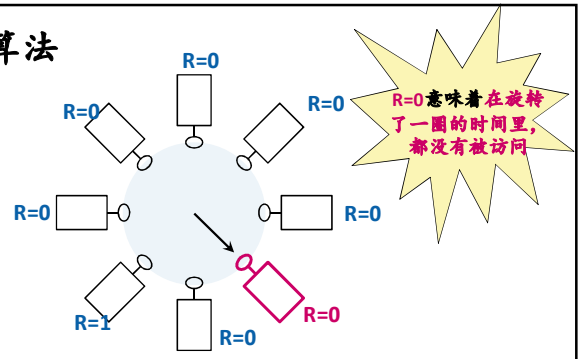
- 替换，顺时针旋转，依次查看下一个页
- if (R == 1) then {R=0; 继续旋转;}

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



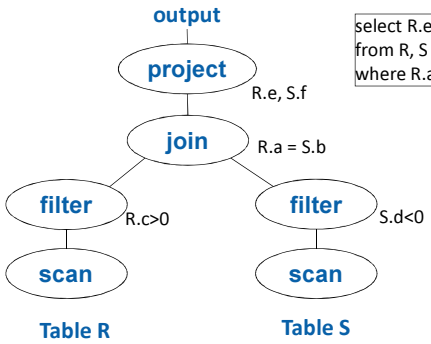
- 替换，顺时针旋转，依次查看下一个页
- if (R == 1) then {R=0; 继续旋转;}
- if (R == 0) then 选中为Victim

Outline

- 数据库系统架构
- 数据存储与访问
  - 数据表
  - 索引
  - 缓冲池
- 运算的实现
  - Operator tree
  - Selection & Projection
  - Join

Operator Tree

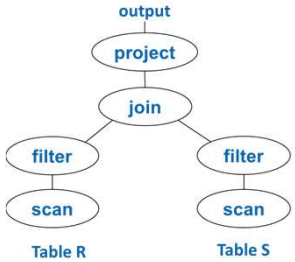
- Query plan 最终将表现为一棵Operator Tree




```
select R.e, S.f
from R, S
where R.a = S.b and R.c > 0 and S.d < 0;
```


Operator Tree

- Query plan 最终将表现为一棵Operator Tree
  - 每个节点代表一个运算
  - 运算的输入来自孩子节点
  - 运算的输出送往父亲节点



- 实现方法
  - Operator at a time
    - 完全处理一个运算再处理下一个运行，会产生大量中间结果
  - Pull（Tuple at a time）
    - 每个Operator实现Open, Close, GetNext方法
    - 父节点调用子节点的GetNext() 取得下一个子节点的输出
  - Push：多线程
    - 子节点把输出放入中间结果缓冲，然后通知父节点去读

Selection & Projection

- Selection: 行的过滤
  - 支持多种数据类型：数值类型，字符串类型等
  - 实现比较操作、数学运算、逻辑运算
- Projection: 列的提取
  - Query plan生成时，同时产生中间结果记录的schema
  - 主要功能：从一个记录中提取属性，生成一个结果记录

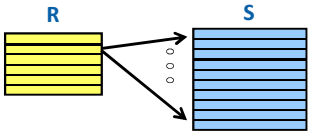
Join的实现

- 三种思路
  - Nested loop
  - Hashing
  - Sorting

Nested Loop Join

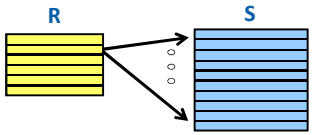
$R \bowtie_{R.a = S.b} S$

```
foreach tuple r ∈ R {
    foreach tuple s ∈ S {
        if (r.a = s.b) output(r,s);
    }
}
```



Nested Loop Join性能分析

$R \bowtie_{R.a = S.b} S$

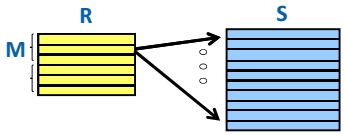


R有 M_R个Page
S有 M_S个Page
每个Page有B个记录

- 外循环读R
  - 读了一遍R
- 内循环读S
  - 对于R的每一个记录读所有的S
  - 总共读了BM_R遍S
- 总共读的page数： $M_R + BM_RM_S$

Block Nested Loop Join

$R \bowtie_{R.a = S.b} S$



内存大小为M

外循环每次读入M页的R，
而不是一条R的记录

内循环读一遍S

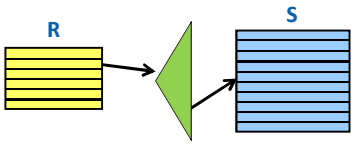
- 外循环读R
  - 读了一遍R
- 内循环读S
  - 总共读了M_R/M遍S
- 总共读的page数： $M_R + M_R M_S/M$

有进步，但仍然是2次的！

Index Nested Loop Join

$R \bowtie_{R.a = S.b} S$

```
foreach tuple r ∈ R {
    lookup index to look for match s in S
    if (found) output(r,s);
}
```

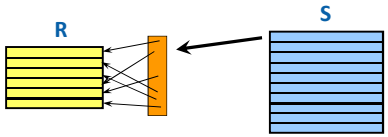


经常被使用，尤其是当很少有匹配时，效率很高

Hash Join

$R \bowtie_{R.a = S.b} S$

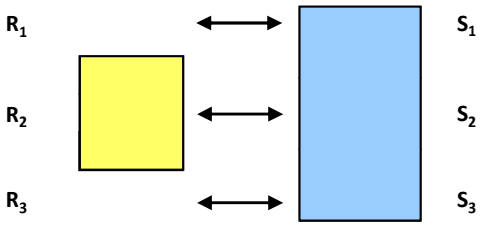
- Simple hash join



读R建立hash table;
读S访问hash table找到所有的匹配;

☞R比内存大怎么办？

I/O Partitioning



- 思路：把R和S划分成小块
  - PartitionID = hash(join key) % PartitionNumber
- R_j中记录的匹配只存在于相应的S_j中
  - 为什么？ 匹配的记录hash (join key)必然相同

GRACE Hash Join

$R \bowtie_{R.a = S.b} S$

- 对R进行I/O partitioning
- 对S进行I/O partitioning
- for (j=0; j< ParitionNumber; j++) {
  simple hash join计算$R_j \bowtie S_j$;
  }

GRACE Hash Join性能分析

R有 M_R个Page
S有 M_S个Page
每个Page有B个记录

- 对R进行I/O partitioning
  - 读M_R个Page，写M_R个Page
- 对S进行I/O partitioning
  - 读M_S个Page，写M_S个Page
- Simple hash join计算所有的$R_j \bowtie S_j$
  - 读$M_R + M_S$个Page
- 总代价（不考虑输出）
  - 读$2M_R + 2M_S$个Page，写$M_R + M_S$个Page

I/O成本是线性的！

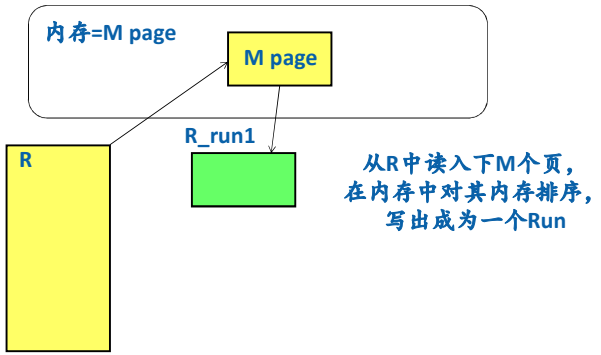
Sort Merge Join

$R \bowtie_{R.a = S.b} S$

- 思路
  - 如果把R按照R.a的顺序排序
  - 如果把S按照S.b的顺序排序
  - 那么可以Merge(归并)找出所有的匹配

Run Generation

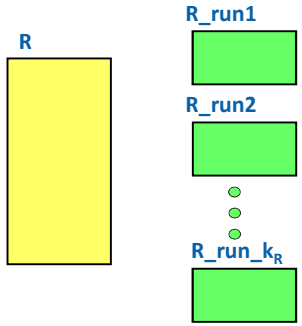
$R \bowtie_{R.a = S.b} S$



从R中读入下M个页，
在内存中对其内存排序，
写出成为一个Run

Run Generation

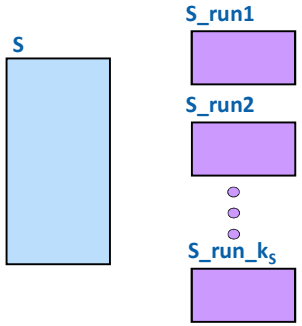
$R \bowtie_{R.a = S.b} S$



$k_R = M_R / M$

Run Generation

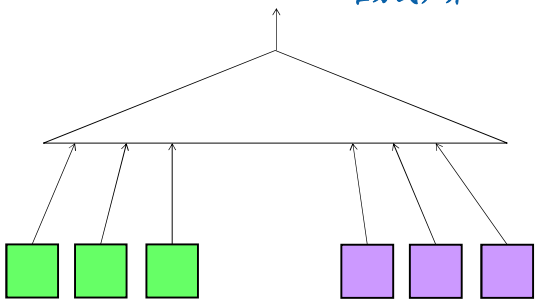
$R \bowtie_{R.a = S.b} S$



$k_S = M_S / M$

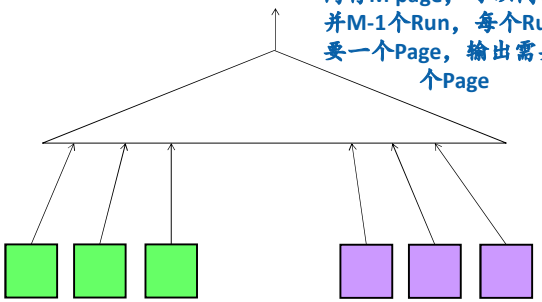
Merge

堆方式归并



Merge

内存M page，可以同时归并M-1个Run，每个Run需要一个Page，输出需要一个Page



需要多少层归并？

- 共有$\frac{M_R}{M}+\frac{M_S}{M}$个Run
- 所以需要$\log_{M-1}(\frac{M_R}{M}+\frac{M_S}{M})$层才能完成全部归并
- 另一个角度：
  - 如果希望只使用一次归并
  - $\log_{M-1}(\frac{M_R}{M}+\frac{M_S}{M})<=1$
  - 那么：$M_R+M_S<=M(M-1)\approx M^2$

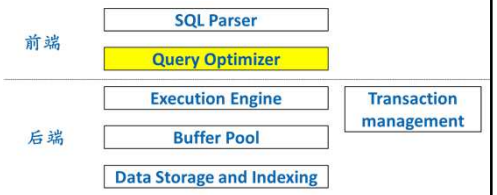
Sort Merge Join

$R\bowtie_{R.a=S.b}S$

- 比较
  - 通常代价比Hash Join稍差
  - 当一个表已经有序的情况下，会被使用

Query Optimization(查询优化)

- 依据
  - 统计信息
- 优化内容
  - 访问方式：顺序扫描？索引？
  - 采用哪种算法
  - 多个连接的先后次序
  - 等等



小结

- 数据库系统架构
- 数据存储与访问
  - 数据表
  - 索引
  - 缓冲池
- 运算的实现
  - Operator tree
  - Selection & Projection
  - Join