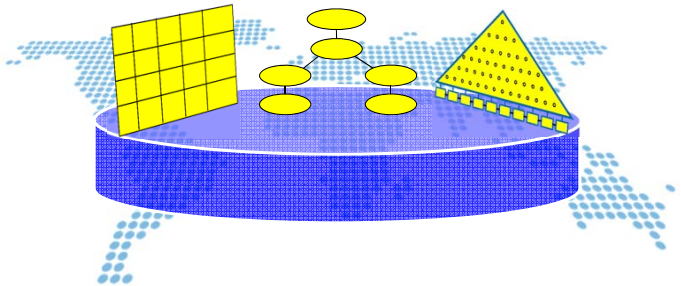
数据库系统

数据存储与访问路径(2)

陈世敏

（中科院计算所）



上节内容

- 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

Outline

- 索引的概念
- 树结构索引
- 哈希索引
- 其他索引

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

☞如果有100个专业会怎么样？

数据的顺序访问

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

☞如果有100个专业会怎么样？

有些浪费

如果每个专业的学生人数大致相同

那么会扔掉99%的记录！

Selective Data Access (有选择性的访问)

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

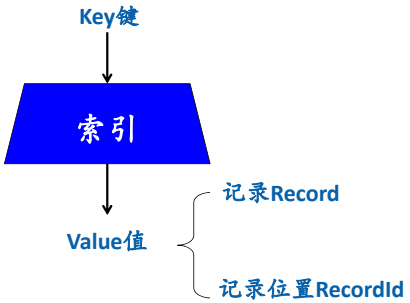
☞如果有100个专业会怎么样？

希望可以直接找到相关的记录

而不需要扫描整个的Table

👌使用索引

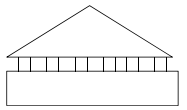
索引（Index）概念



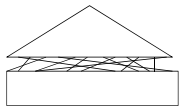
- 给定一个键，找到对应的值
- 在数据库里，值通常对应于记录或记录的位置

索引的种类

- 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/主索引（Clustered Index, primary index）：顺序一致
    - 每个表上只有一个
  - 二级索引（Non-clustered index, Secondary Index）：顺序不一致



Clustered

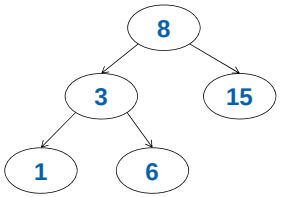


Non-Clustered
又称作Secondary

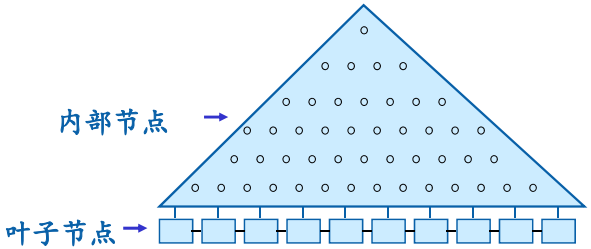
Outline

- 索引的概念
- 树结构索引
  - B+-Tree
  - 索引访问代价
  - 内存优化的B+-Tree
- 哈希索引
- 其他索引

二叉查找树?

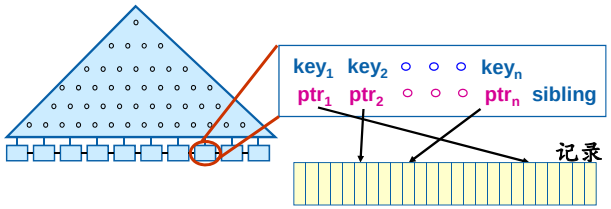


B+-Trees



- **是二叉树的推广**
  - 每个节点可以有多个孩子
  - 永远是平衡的
- 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

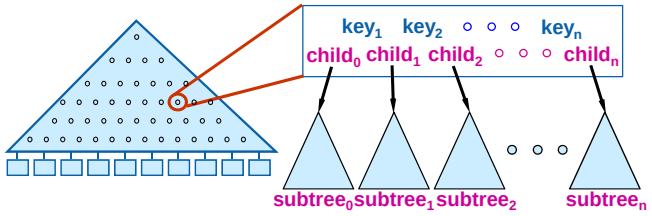


所有叶子逻辑上组成一个数组

叶节点自左向右从小到大排序，以sibling pointer链起来
(ptr= record ID; sibling = page ID)

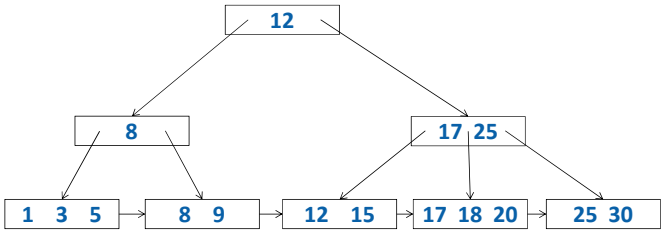
节点内Keys 从小到大排序: $key_1 < key_2 < \cdots < key_n$

内部节点



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

举例

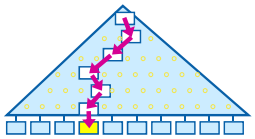


假设每个节点的child/pointer个数最多为B=3

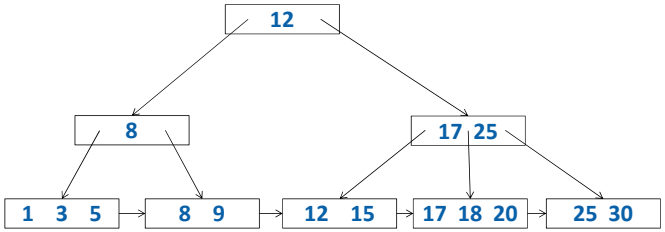
B+-Tree: Search

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

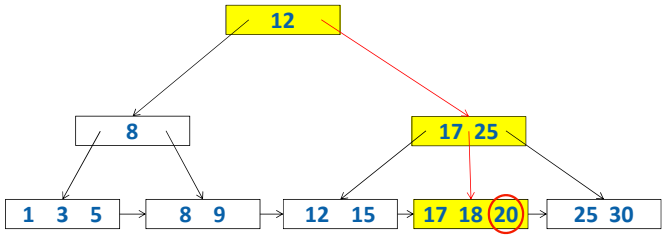


举例 Search(20)



假设每个节点的child/pointer个数最多为B=3

举例 Search(20)

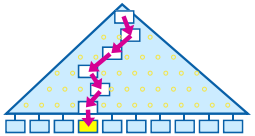


假设每个节点的child/pointer个数最多为B=3

B+-Tree: Search

Search代价

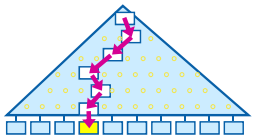
- 共有N个key
- 每个节点的child/pointer个数为B
- 树高：$O(\log_B N)$
- 总比较次数
  - 每个节点内部二分查找：$O(\log_2 B)$
  - $O(\log_B N) \times O(\log_2 B) = O(\log_2 N)$
- 总I/O次数= $O(\log_B N)$



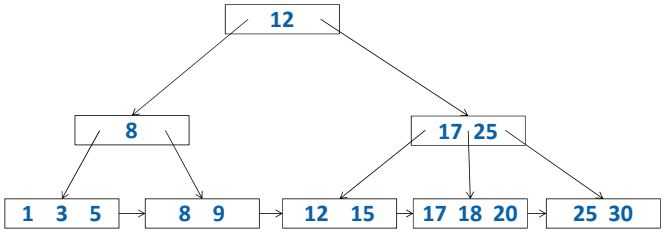
B+-Tree: Insertion

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)



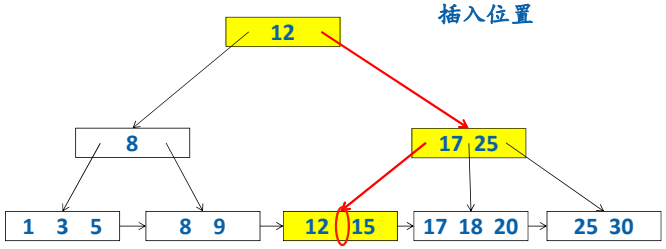
举例 Insert(14)



假设每个节点的child/pointer个数最多为B=3

举例 Insert(14)

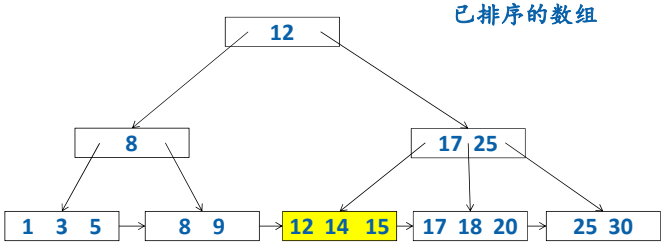
通过search找到插入位置



假设每个节点的child/pointer个数最多为B=3

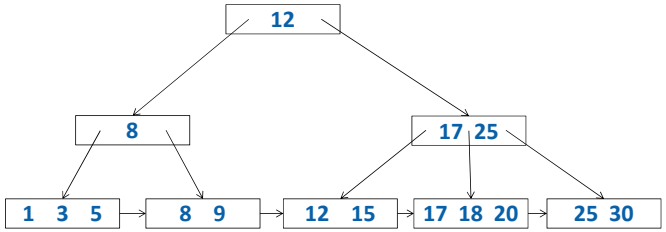
举例 Insert(14)

叶子未满，插入已排序的数组



假设每个节点的child/pointer个数最多为B=3

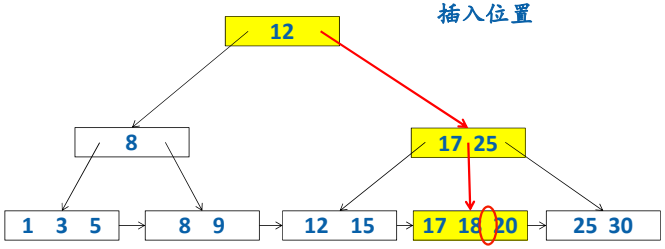
举例 Insert(19)



假设每个节点的child/pointer个数最多为B=3

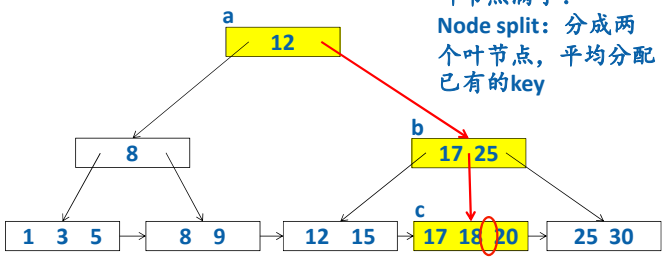
举例 Insert(19)

通过search找到插入位置



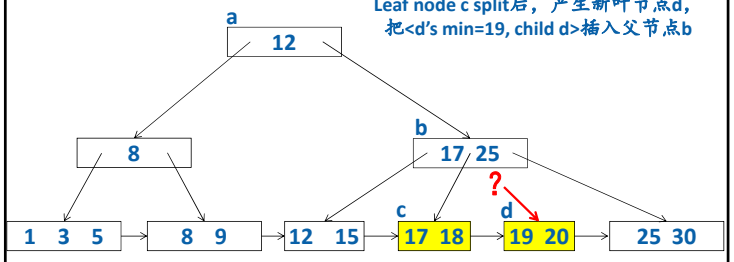
假设每个节点的child/pointer个数最多为B=3

举例 Insert(19)



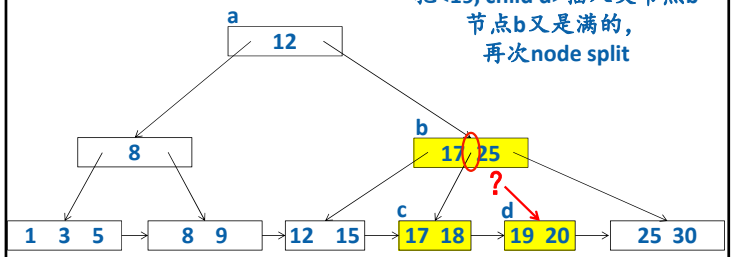
假设每个节点的child/pointer个数最多为B=3

举例 Insert(19)



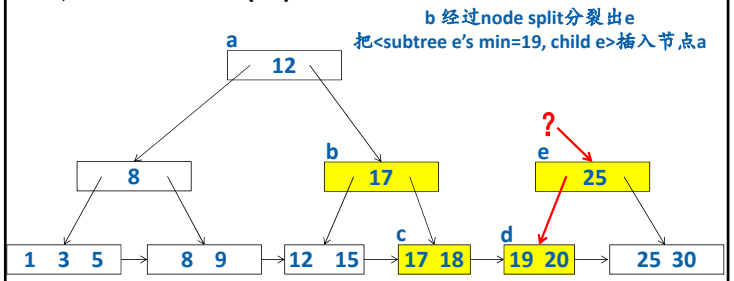
假设每个节点的child/pointer个数最多为B=3

举例 Insert(19)



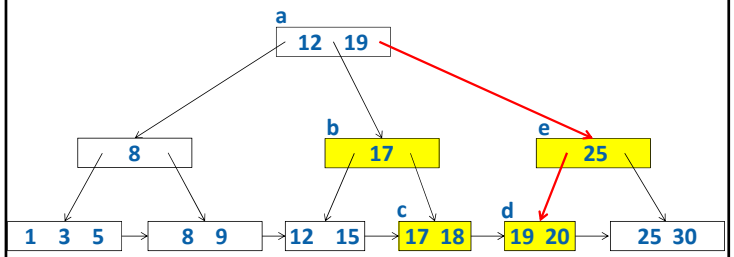
假设每个节点的child/pointer个数最多为B=3

举例 Insert(19)



假设每个节点的child/pointer个数最多为B=3

举例 Insert(19)

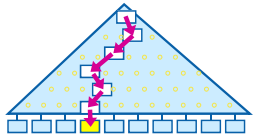


假设每个节点的child/pointer个数最多为B=3

B+-Tree: Deletion

Deletion

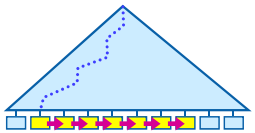
- Search 然后在节点中删除
- node merge?
  - 原设计：当节点中key个数小于一半
  - 实际实现：数据总趋势是增长的
    可以只有节点为空时才node merge
    或者完全不进行node merge



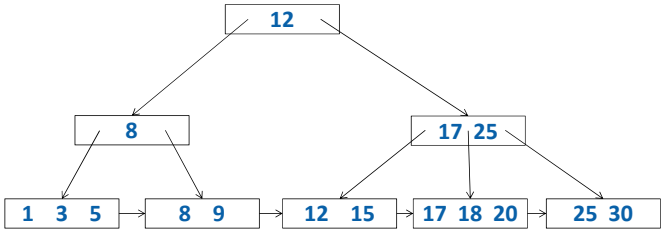
B+-Tree： Range Scan

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

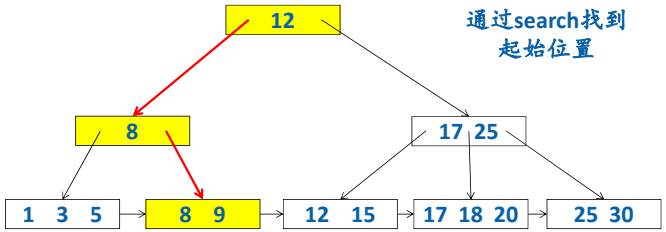


举例 Range scan (9, 20): 获取[9, 20]区间的index entry



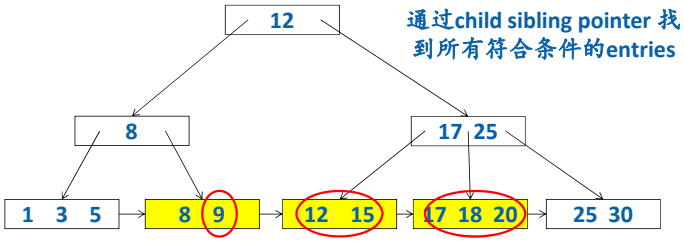
假设每个节点的child/pointer个数最多为B=3

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



假设每个节点的child/pointer个数最多为B=3

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

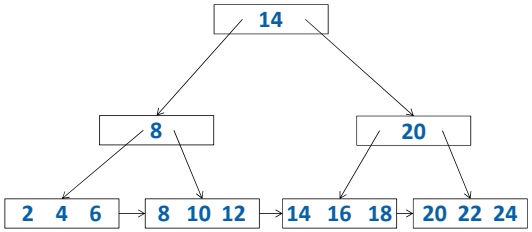


假设每个节点的child/pointer个数最多为B=3

练习1

- 假设每个节点的child/pointer个数最多为B=3
- 树中包括下述key:
  2,4,6,8,10,12,14,16,18,20,22,24
- 要求
  - 节点中尽量放满
  - 内部节点至少有2个孩子
- 请画出B+-Tree

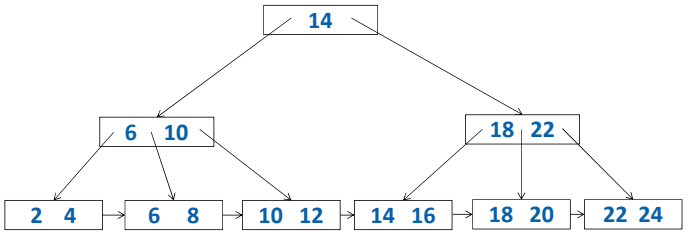
练习1解答



练习2

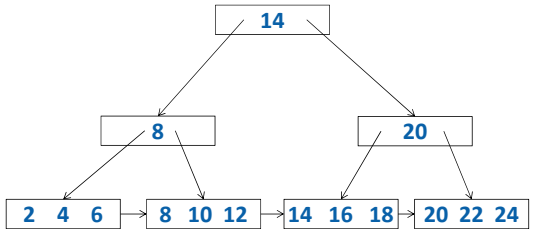
- 假设每个节点的child/pointer个数最多为B=3
- 树中包括下述key:
  2,4,6,8,10,12,14,16,18,20,22,24
- 要求
  - 叶节点保留至少1个空位
  - 内部节点至少有2个孩子
- 请画出B+-Tree

练习2解答

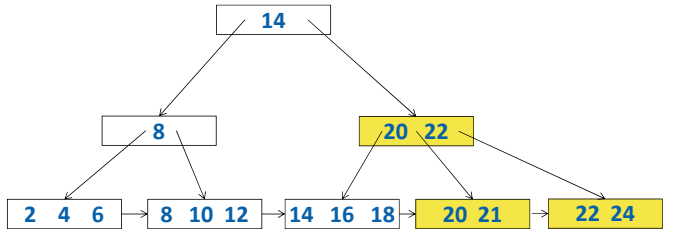


练习3

- 假设每个节点的child/pointer个数最多为B=3
- 请画出insert(21)之后的B+-Tree



练习3解答



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
- 假设有K个匹配的tuple
  - 那么对于数据将进行K次随机读

比较顺序访问与二级索引访问

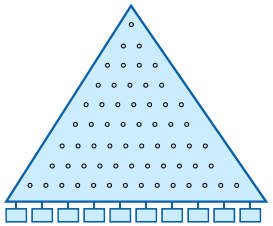
- 顺序访问
  - 需要处理每一个记录
  - 顺序读每一个page
- 二级索引访问
  - 有选择地处理记录
  - 随机读相关的page

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

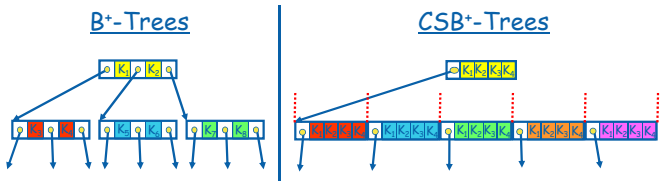
内存优化的B⁺-Trees

- Main memory B⁺-Trees
- Node width = cache line size



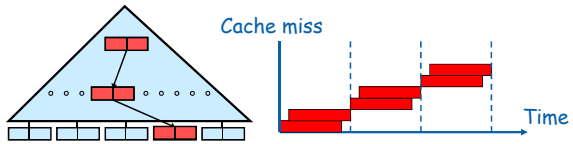
Reducing Pointers for Larger Fanout

- Cache Sensitive B⁺-Trees (CSB⁺-Trees) [RR00]
- Layout child nodes contiguously
- Eliminate all but one child pointers
  - *Double fanout of nonleaf node*

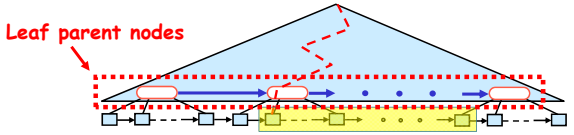


Prefetching for Larger Nodes

- Prefetching B+-Trees [CGM01]
- Node size = multiple cache lines (e.g. 8 lines)



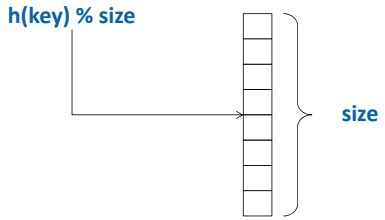
Prefetching for Range Scan Performance



Outline

- 索引的概念
- 树结构索引
- 哈希索引（散列表）
  - Hash function
  - Chained hashing
  - Extendible hashing
- 其他索引

哈希表（Hash Table）



- 思路：索引查找➔地址/下标运算
- 哈希函数h( )?
- 冲突解决？

哈希函数h( )

- 目的：键key➔近乎随机的整数
- 通常对key的字节串进行运算

乘积型哈希函数h( )

```
uint32_t multhash(const char *key, int len) {
   uint32_t hash = INIT_VAL;
   for (uint32_t i = 0; i < len; ++i)
      hash = M * hash + key[i];
   return hash;
}
```

(例如： Kernighan and Ritchie's function, INIT_VAL=0, M=31)

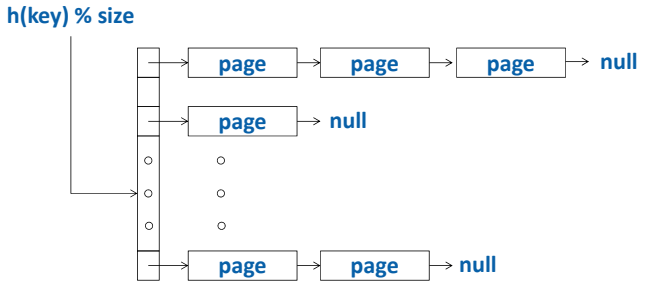
复杂哈希函数举例

```
uint32_t jenkins_one_at_a_time_hash(char *key, size_t len)
{
    uint32_t hash, i;
    for(hash = i = 0; i < len; ++i)
    {
        hash += key[i];
        hash += (hash << 10);
        hash ^= (hash >> 6);
    }
    hash += (hash << 3);
    hash ^= (hash >> 11);
    hash += (hash << 15);
    return hash;
}
```


冲突解决方法？

- 注意
  - 可能h(key1) == h(key2)
  - 可能h(key1)%size == h(key2) %size
- 数据结构课上
  - 线性散列等
- 常见的方法：链表/溢出Page的方法

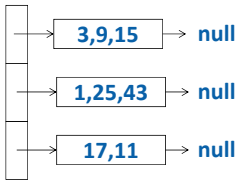
Chained Hash Table



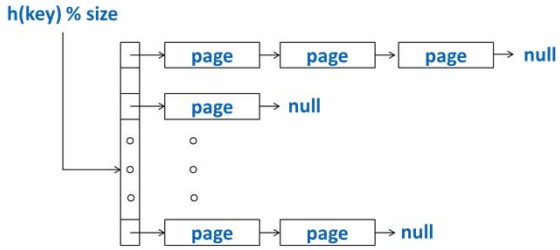
- 每个bucket由1到多个Page组成

举例

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

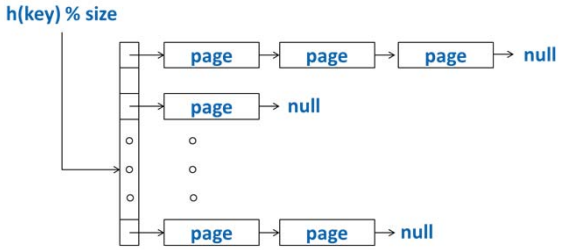


当链表太长时性能变差



- 平均长度怎么计算？
  - (总索引项个数/每页索引项个数)/size

简单方法：Rehashing



- 建一个新的Hash Table代替原来的
- 把Table_Size变大： Table_size *= 2;
- 代价昂贵！

为了减少Rehashing的代价

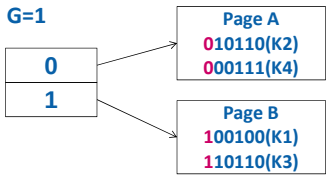
- 有两种经典的动态方案
- Extendible Hashing（可扩展哈希）
  - 我们这里介绍
- Linear Hashing（线性哈希）
  - 如果有兴趣，可以自学

Extendible Hashing

- 基本目标
  - 能不能把rehashing的代价变小？
  - 尽量减少需要移动的索引项
- 基本想法
  - 采用一种特殊的哈希函数
  - 使size加倍后，原有的bucket和新的bucket有某种对应关系

Extendible Hashing

```
h(k1)=100100
h(k2)=010110
h(k3)=110110
h(k4)=000111
```



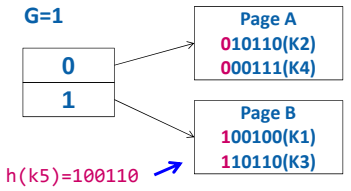
假设每个Page只能放2个项

- bucket下标=hash后的整数的前G位
  - G: Global depth
  - Table size = 2^G

Extendible Hashing

```
h(k1)=100100
h(k2)=010110
h(k3)=110110
h(k4)=000111
```

h(k5)=100110 怎么办？



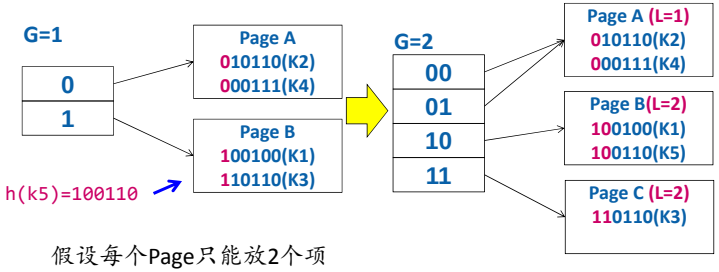
假设每个Page只能放2个项

- bucket下标=hash后的整数的前G位
  - G: Global depth
  - Table size = 2^G

Extendible Hashing

h(k1)=100100 h(k5)=100110
h(k2)=010110 怎么办?
h(k3)=110110
h(k4)=000111

- bucket下标=hash后的整数的前G位
  - G: Global depth
  - Table size = 2^G

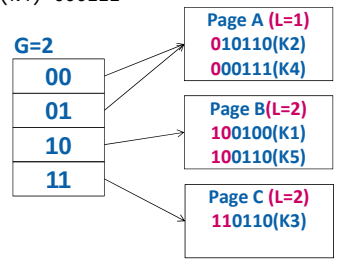


假设每个Page只能放2个项

Extendible Hashing

h(k1)=100100 h(k5)=100110
h(k2)=010110 怎么办?
h(k3)=110110
h(k4)=000111

- bucket下标=hash后的整数的前G位
  - G: Global depth
  - Table size = 2^G
- 变化
  - G=G+1，Table Size变为2倍
  - 每个原来的Bucket对应于现在两Bucket
  - B分裂为了2个
  - A有两个bucket指针指向
- 这个时候出现了不同种的Page，我们用L表示
  - L: Local depth
  - 每个Page中前L位相同

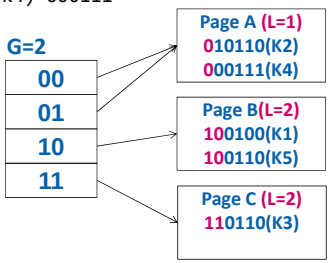


假设每个Page只能放2个项

Extendible Hashing

h(k1)=100100 h(k5)=100110
h(k2)=010110 h(k6)=000000
h(k3)=110110
h(k4)=000111

- bucket下标=hash后的整数的前G位
  - G: Global depth
  - Table size = 2^G

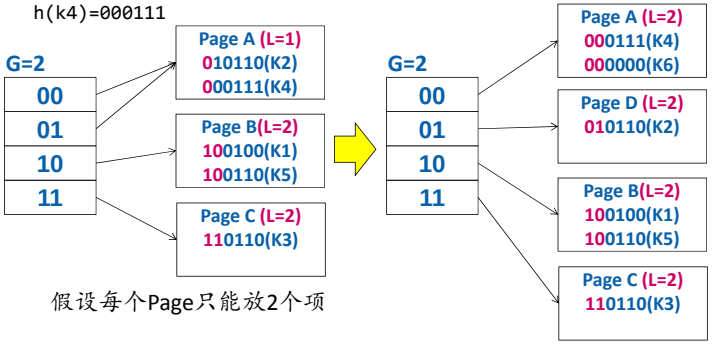


假设每个Page只能放2个项

Extendible Hashing

h(k1)=100100 h(k5)=100110
h(k2)=010110 h(k6)=000000
h(k3)=110110
h(k4)=000111

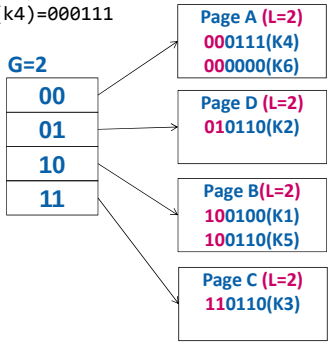
- bucket下标=hash后的整数的前G位
  - G: Global depth
  - Table size = 2^G



假设每个Page只能放2个项

Extendible Hashing

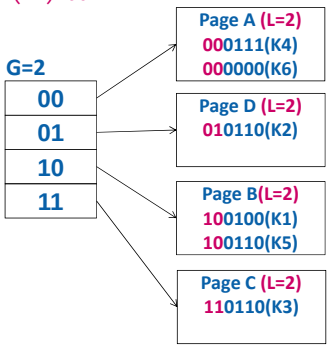
h(k1)=100100 h(k5)=100110
h(k2)=010110 h(k6)=000000
h(k3)=110110
h(k4)=000111



- bucket下标=hash后的整数的前G位
  - G: Global depth
  - Table size = 2^G
- A分裂为A和D
  - 但是G不需要变化
- 如何区分这两种情况呢？
  - 分裂页的L==G，需要G=G+1，加倍table size
  - 分裂页的L<G，只需要分裂，不需要改变table size

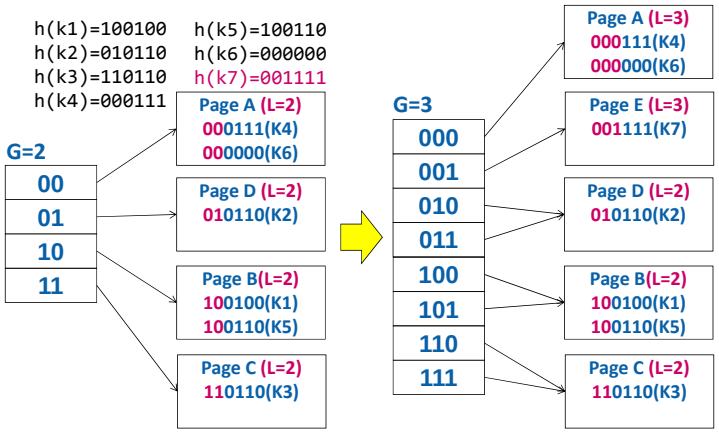
Extendible Hashing

h(k1)=100100 h(k5)=100110
h(k2)=010110 h(k6)=000000
h(k3)=110110 h(k7)=001111
h(k4)=000111



Extendible Hashing

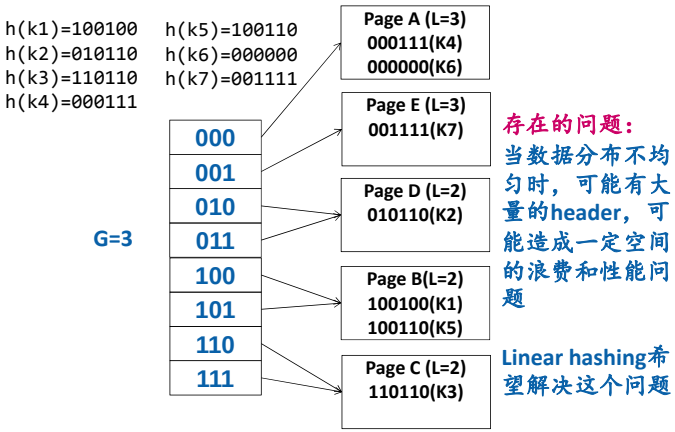
h(k1)=100100 h(k5)=100110
h(k2)=010110 h(k6)=000000
h(k3)=110110 h(k7)=001111
h(k4)=000111



Extendible Hashing：Insert(key)

```
hash= h(key);
page= table[prefix(hash, G)];
if (page不满) {
  把 key , hash 插入Page;
}
else {
  if (page.L == G) {
    G++, 把Table[]变为原来的两倍
    Table'[i<<1|0] = Table'[i<<1|1] = Table[i];
  }
  page.L ++;
  分配一个新的page, 根据prefix(*, page.L)分配各项;
  修改相应的Table[]内容;
}
```

Extendible Hashing



存在的问题：
当数据分布不均匀时，可能有大量的header，可能造成一定空间的浪费和性能问题

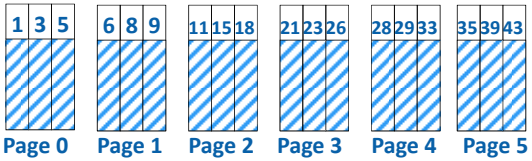
Linear hashing希望解决这个问题

Outline

- 索引的概念
- 树结构索引
- 哈希索引
- 其他索引
  - ISAM
  - 位图索引
  - 倒排索引

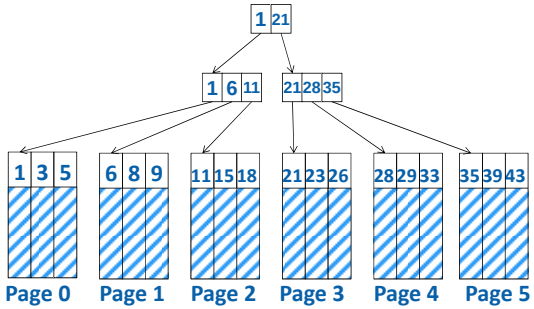
排序文件

- 按照某个列的顺序排序
  - 还记得无序的文件叫什么吗？
- 可以二分查找
  - 有什么问题？
  - 每次需要一次I/O读一个Page



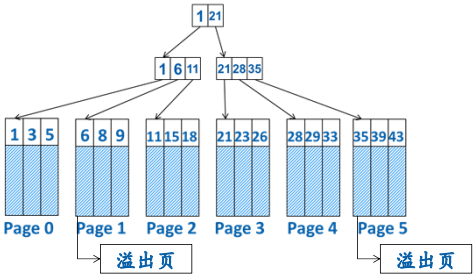
ISAM：排序文件+多层索引

- 在排序文件上建立多层索引
  - 紧凑不留空位



这种索引又称为ISAM (Indexed Sequential Access Method)

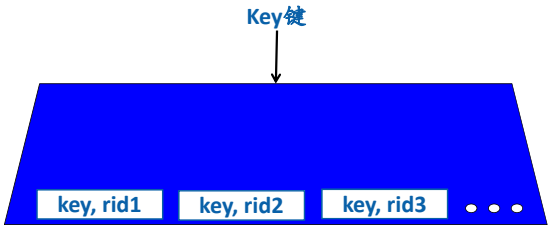
ISAM插入新记录怎么处理？



- 上面索引部分不变
- 数据部分增加溢出页
  - 但是溢出页增多，会影响性能

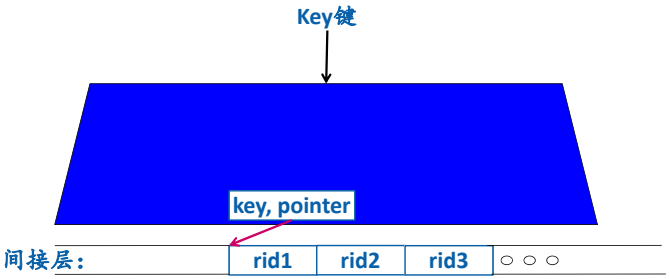
所有索引实际上都需要支持：
相同Key有多个记录？

方法1：直接作为不同的项



- 在树结构或哈希表中允许多项具有相同的key
- 问题：当有很多项具有相同的key时，
  key在索引中存在了多份，占用了额外的空间

方法2：间接层



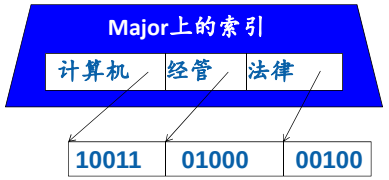
- 在树结构或哈希表中只记录唯一的key
- 多个RecordId记录在间接层中

方法3：Bitmap Index位图索引

- 把间接层记录为多个位图
- 每个不同的key对应一个bitmap
  - bitmap中的每位对应于一条记录
  - 1表示：这条记录在这列上取值==key
  - 0表示：这条记录在这列上取值!=key
- 适用范围

Bitmap Index位图索引

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



位图索引分析

- 位图的大小与key的个数成正比
  - 索引列的取值个数越多，位图索引需要的位图数越多
- 适用场合：当索引列取值个数少的情况
  - 例如：性别、专业、年级等
  - 但是不适合：姓名

位图的优势

- 过滤条件的求值可以通过位运算来实现
- Column = value
  - 从Column对应的索引中得到value所对应的位图
- Column >= value
  - 从Column对应的索引中得到所有大于value所对应的位图
  - 把这些位图OR起来
- 条件1 and 条件2
  - 两个位图的AND
- 条件1 or 条件2
  - 两个位图的OR

位图的压缩

- 当1的个数非常少时，可以进行压缩
- 主要思想
  - 记录两个1之间的距离：表示A个0后跟一个1
  - 采用某种编码，用变长整数表达距离A
    - 关键问题是如何表示长度？
- 在使用时，解压进行处理
  - 位运算是逐位顺序进行的
  - 可以解压一段计算一段

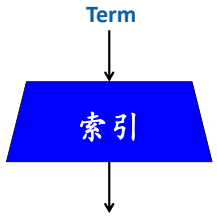
Inverted Index（倒排索引）

- 主要用于对文本进行索引
  - 有大量的文档（Document）
  - 每个文档有很多单词（Term）组成

  Doc1: {t1, t3, t4, …, t1000}
  Doc2: {t2, t3, t4, …, t9999}
  Doc3: {t5, t3, t9, …, t8888}
  ……

- 目标
  - 给定关键字term
  - 找到相关的文档，并对结果排序

倒排索引



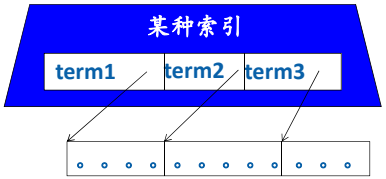
- 在搜索引擎中广泛使用
- 对于关系型数据库系统：可以对文本列产生倒排索引，从而加速like等操作

让我们来分析一下需求

- Key的个数：可能的term(单词，名词等)有多少个？
  - 比较少
  - 常见的词汇：~几万，~几十万？
- 文档有多少个
  - 搜索引擎网页数量
  - 大概450亿
- 同一个key有大量的记录

数据来源：http://www.worldwidewebsize.com/

在term上很容易建立索引



- 可以是B+-tree, hash table等
- 主要的难点：
  如何记录每个term对应的大量的DocID和TF？

Inverted List（倒排表）

Term->($DocID_1$,TF_1), ($DocID_2$,TF_2), …, ($DocID_n$,TF_n)

- 这个结构被称为Inverted List，连续存储于外设
  - 可以用顺序读访问
- 设计的重要问题是如何对DocID和TF进行压缩
  - DocID从小到大排序，只记录相邻两项DocID的差值
  - 采用恰当的变长整数编码

小结

- 索引的概念
- 树结构索引
  - B+-Trees
  - 索引访问代价
  - 内存优化的B+-Tree
- 哈希索引
  - Hash function
  - Chained hashing
  - Extendible hashing
- 其他索引
  - ISAM
  - 位图索引
  - 倒排索引