数据库技术-数据存储与索引

赵亚伟

zhaoyw@ucas.ac.cn

中国科学院大学 大数据分析技术实验室

2017.12.3

目录

深入DBMS

- 利用约3次-4次课深入到DBMS内部来研究数据库技术。
- 对于DBA和开发人员来说，如何设计良好的数据库、写出高效的表达式以及采用什么形式的索引都需要深入DBMS来了解其运行机制。
- 三个研究内容
  - 数据存储与索引
  - 查询处理及优化
  - 事务管理、并发控制及恢复系统

为什么要研究这三个内容？

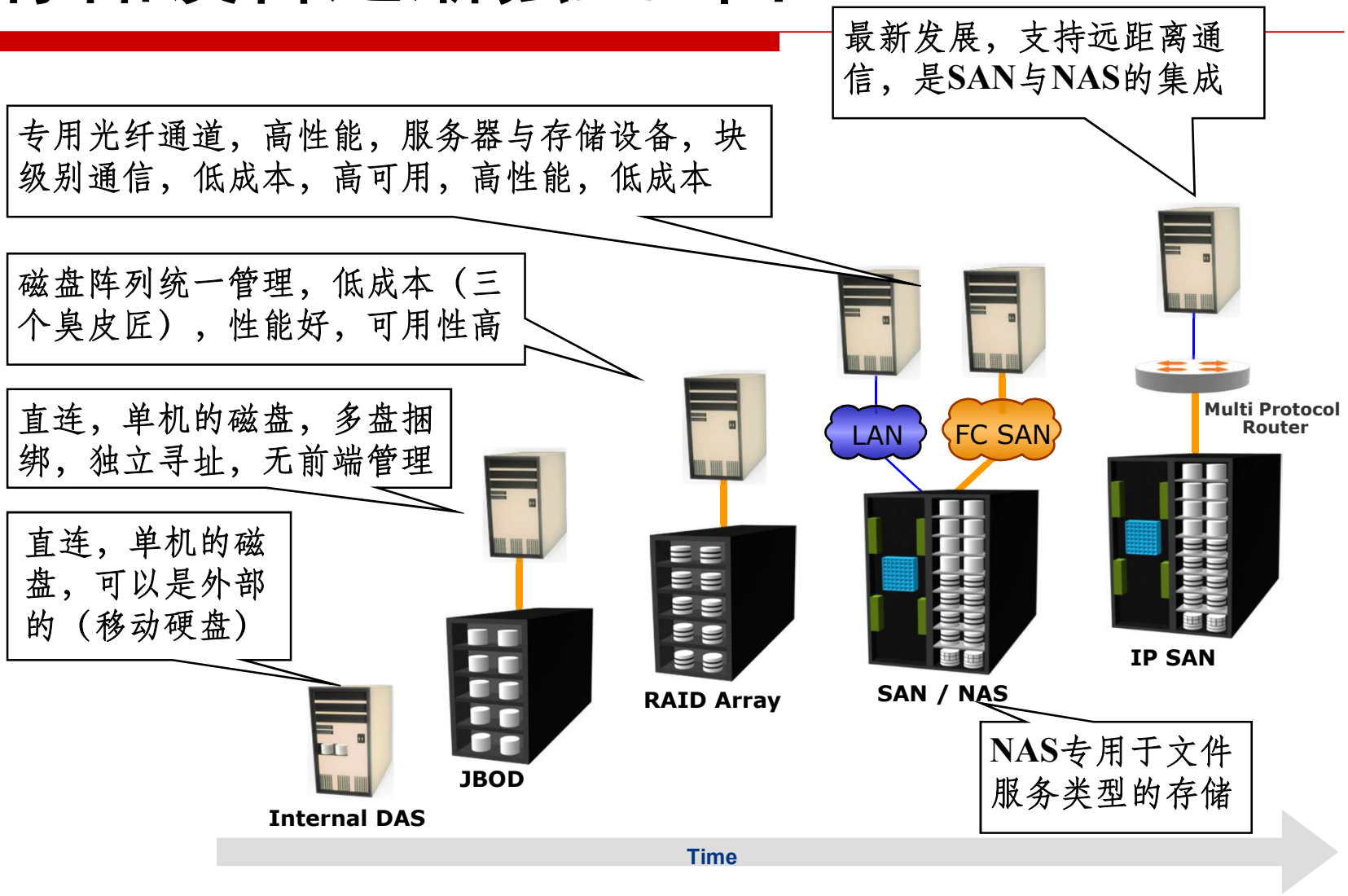
- 最直接的三个问题：
  - 建立什么样的索引能够提高查询效率？存储及索引
  - 如何调整查询表达式才能得到最好的查询效果？查询处理及优化
  - 如何优化事务以提高处理的效率，如何应对错误及故障？事务处理及并发控制，恢复系统
- 对于大型数据库，上述三个问题是必须面对的，也是必须解决的。
- 解决这三个问题就需要深入到**DBMS**内部分析。

目录

存储技术和架构发展

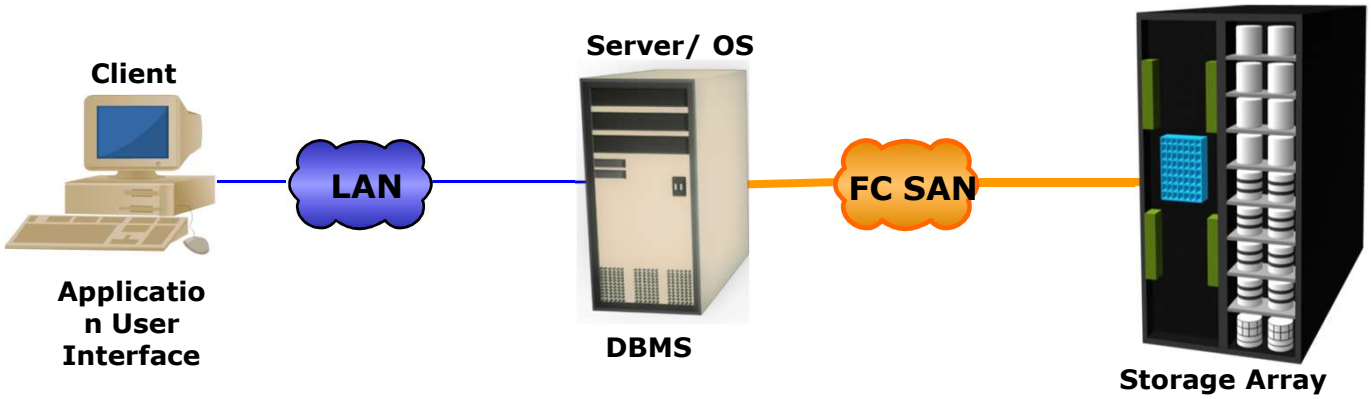
- 数据中心：集中计算机（大型机）和信息存储设备（磁带卷和磁盘架）
- 分散的企业部门内部服务器导致信息难于维护、不易管理，信息孤岛，增加开销。
- 架构发展：非智能存储→智能存储，存储独立，海量存储
  - 冗余磁盘阵列（**Redundant Array of Independent Disks，RAID**）
  - 直连存储（**Direct-attached storage，DAS**）
  - 简单磁盘捆绑（**Just a Bunch Of Disks，JBOD**）
  - 存储区域网（**Storage area network，SAN**）
  - 网络互联存储（**Network-attached storage，NAS**）
  - **IP存储区域网（Internet Protocol SAN，IP-SAN）**

存储设备逐渐独立出来


最新发展，支持远距离通信，是SAN与NAS的集成
专用光纤通道，高性能，服务器与存储设备，块级别通信，低成本，高可用，高性能，低成本
磁盘阵列统一管理，低成本（三个臭皮匠），性能好，可用性高
直连，单机的磁盘，多盘捆绑，独立寻址，无前端管理
直连，单机的磁盘，可以是外部的（移动硬盘）
LAN
FC SAN
Multi Protocol Router
IP SAN
SAN / NAS
RAID Array
JBOD
Internal DAS
NAS专用于文件服务类型的存储
Time

核心部件

- 应用：应用系统，部署在数据库之上。
- 数据库：DBMS可以优化存储和检索数据。
- 服务器和操作系统：运行应用系统和数据库的计算平台。
- 网络：介于客户端和服务器之间的数据通路。
- 存储阵列：永久存储数据以供后续使用的设备。



Example of an Order Processing System

关键需求

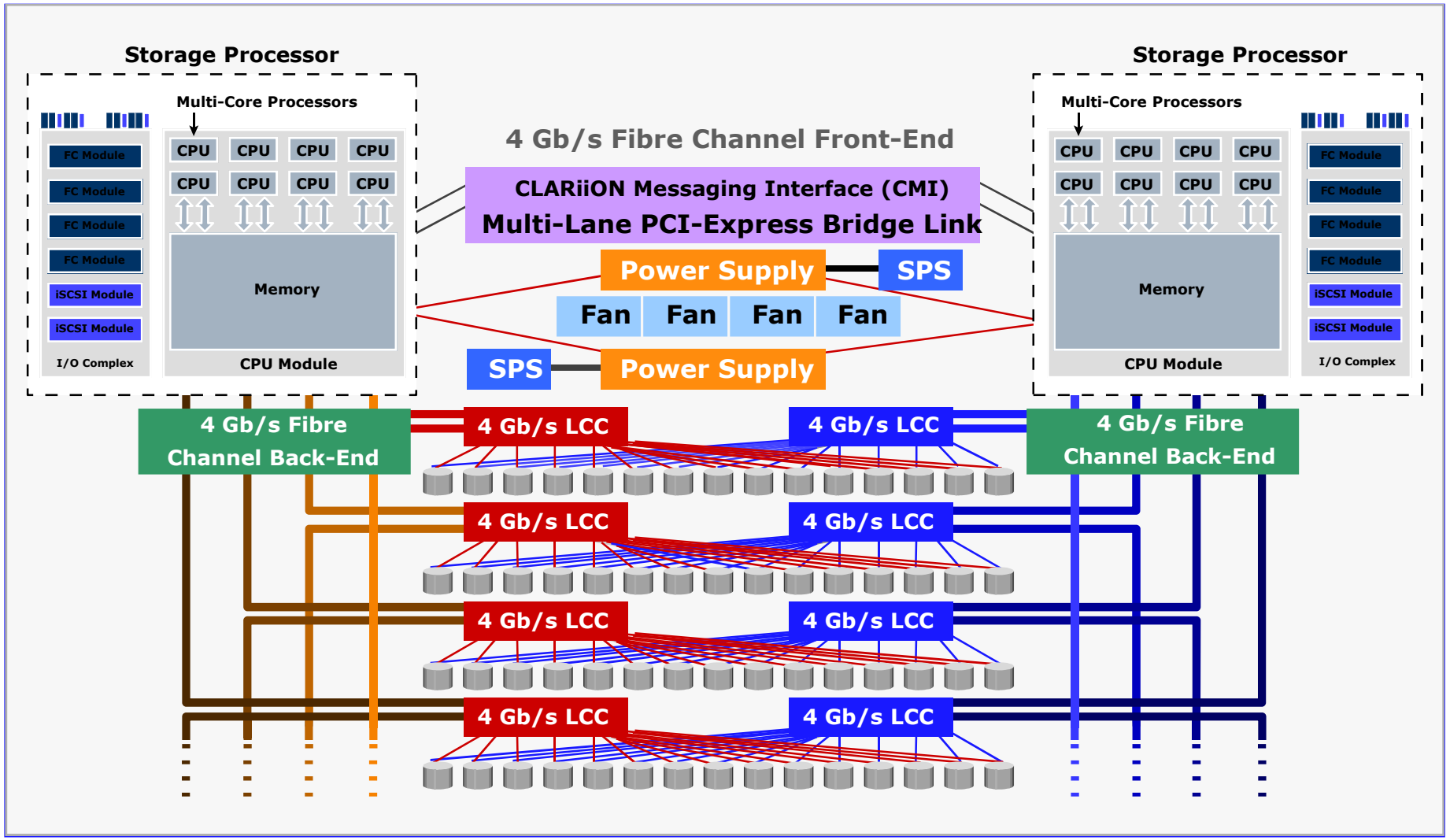
- 数据中心操作的不可中断性对商业机构至关重要，需要满足的关键需求包括：
  - 可用性：随时数据都可以访问
  - 安全性：能够阻止未授权的访问
  - 可扩展性：在不中断商业运营的前提下，按需增加额外的处理能力和存储能力
  - 性能：能够支持高性能的存取操作
  - 数据完整性：不允许存在数据存取偏差
  - 容量：能够高效地存储和处理海量数据
  - 可管理性：可以通过专用的工具管理存储设备

案例：EMC的存储阵列

EMC CLARiiON

EMC Symmetrix

案例：CLARiiON CX-4 Architecture



文件组织

- 数据的物理组织就是要解决如何在存储设备中安排和组织数据以及对数据实施访问的方式。
- 物理组织的主要内容是把有关联的数据组织成一个个的物理文件，故又称之为文件组织，是操作系统中文件系统的扩充。
- **文件组织解决的问题—如何把有关联的数据（记录）组织成文件**
- 把数据库映射到文件中的一种方法是使用多个文件，每个文件只存贮固定长度的记录—定长记录；另一种方法是使用一个文件，使之能容纳多种长度的记录—变长记录

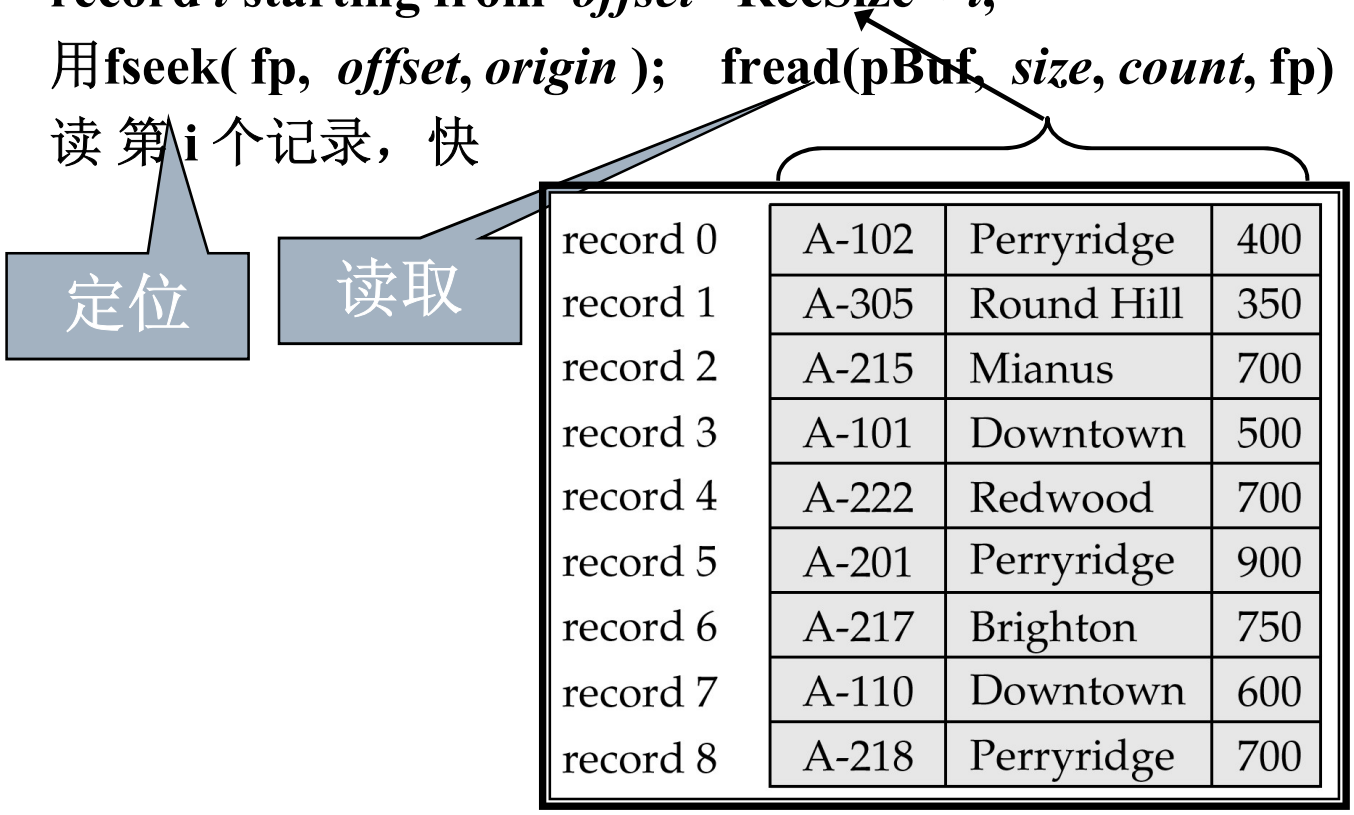
定长记录

- 优点，起止点易计算，不需搜索匹配。
- 定长记录由操作系统解决，已经比较满意。

record *i* starting from *offset*= RecSize * *i*,

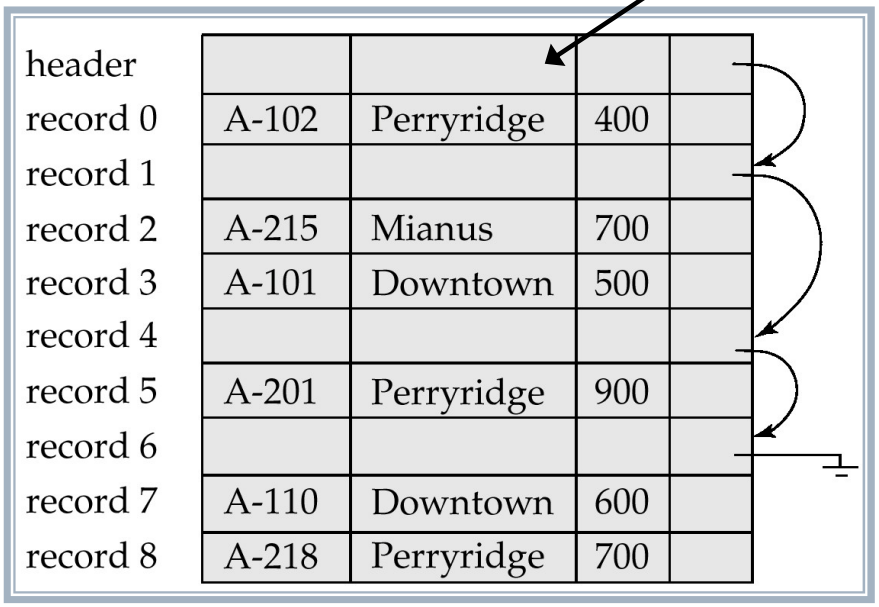
用**fseek(fp, *offset*, *origin*);** **fread(pBuf, *size*, *count*, fp)**

读 第 i 个记录，快



定长记录-空间收回列表

- 删除时不需真删除，只需加一个删除标志，可以后悔。
- 把确认真正删除的记录链接起来，称为回收列表 （**free List)**，以后要插入记录时，依次使用回收列表中的位置



变长记录存储

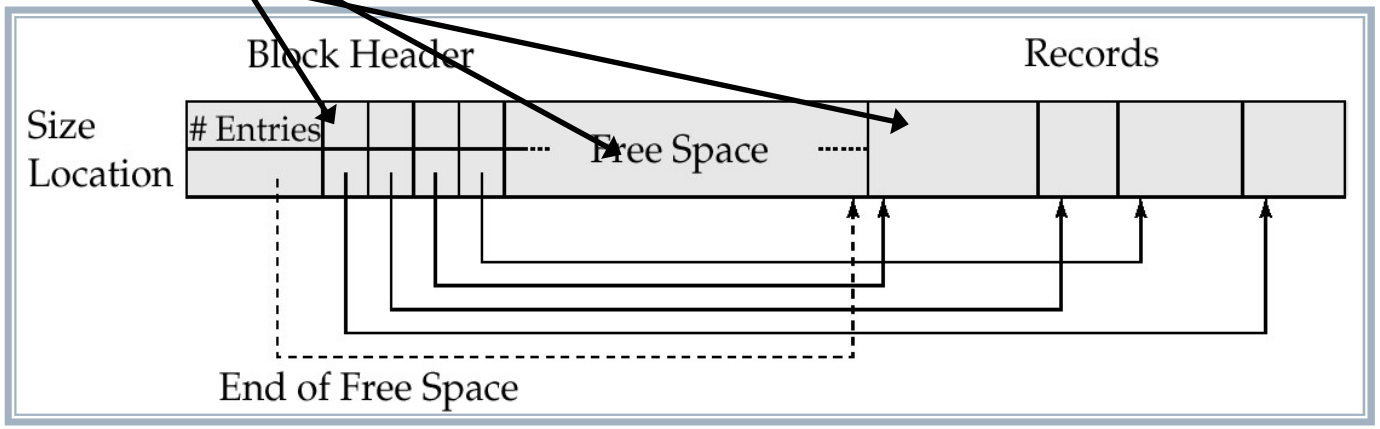
- 变长记录以下面几种方式出现在数据库系统中：
  - 多种记录类型在一个文件中存储
  - 允许一个或多个字段是变长的记录类型（如对象）
  - 允许重复字段的重复的记录类型，如数组、多重集合
- 例如：
  - **1 TXET,** 行长不确定，**\r\n**为行尾
  - **2 Book=(Prefix(chapter(Section)*)*Index)**
  - **3 XML**文件
  - **4** 歌曲磁带，搜索 计算空白数
- 优点：可存储多种类型的数据
- 缺点：删除操作费力，插入操作费力

块

- 操作系统决定了一个块（**block**）的大小，块是磁盘**I/O**请求的基本单位（要么全读，要么全不读），每个**I/O**请求指定要访问的磁盘地址，地址是以“块号”的形式提供的。一个块是一个逻辑单元，包含固定数目的连续扇区。
- 块的大小在**512**字节到几**KB**之间，一般块的大小为**4KB**。不同的**DBMS**对块大小的划分是不同的，如**Oracle**块是**2KB**或**4KB**，而**SQL Server**块是**8KB**。
- 块的大小是固定的，但记录的大小可以不同，在关系数据库中，不同关系中的元组通常有不同的大小

分槽的页结构

- 分槽的页结构 （**Slotted-Page Structure**）一般用于块中组织记录，即将一个块作为一个槽页
- 每个槽页分三个区（类似光盘的目录组织）：
  - 块头区：记录个数、空闲的末尾位置、每条记录的位置及大小（组成数组）
  - 空闲区：连续的，在块头和记录区之间
  - 记录区：连续的，在空闲区后至块尾



插入删除

- 如果一个记录被删除，它所占用的空间被释放，并且它的条目被置成删除状态（如大小置为**-1**），块中被删除记录之前的记录向后移动，使产生空闲空间可以被重新使用，块头的空闲空间指针也做相应修改
- 如果一个记录插入，插入到空闲空间尾部，同时调整块头的相关信息
- 注意：移动记录的代价不会太高，因为块的大小是被限制的，典型的值为**4KB**

小结

- 存储设备逐渐从主机中独立出来，形成数据中心，有专用的工具进行管理。
- 块和页是存储的两个重要概念。
- 文件的物理组织形式与实际需求密切相关，如顺序文件适合某搜索码顺序排序的记录
- 尽管数据存储会影响查询效率，但由**DBMS**处理，索引是提高查询效率的另一种重要技术。

目录

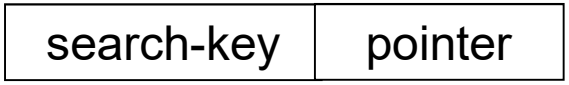
- 问题的提出
- 数据存储
- 索引基本概念
- 有序索引
  - 主索引与辅助索引
  - 稠密索引与稀疏索引
  - 多级索引
  - 索引更新
  - B^+树索引
- 散列
  - 静态散列
  - 动态散列
  - 顺序索引与散列比较
  - 位图索引（自学）

索引–基本思想

- 索引的基本思想：
  - 词典的索引能够帮助迅速找到所需要查询的单词，而词典内容则是对每个单词的解释和说明。
  - 建立索引表是将每页的某一个单词与页号列表对照，那么查单词可先查表(称为索引表)，等确定页面号后，再细查该页面。
  - 索引文件源于上述基本思想，组织索引表(简称索引)是索引文件的关键。
- 例：以年龄为搜索码，定义**Hash(Age) = Age Mod 12**，把人按年龄分成**12**个组，即 通常的**12**属相“鼠、牛、虎、兔，...”。在同一属相中再线性搜索，查询时，先问属相，如属虎，再到“虎”属性中查，寻址效率提高 **12**倍。

索引-基本概念

- 有序索引：基于值的排序顺序建立的索引，又称顺序索引（索引有序）。
- 无序索引：基于将值平均分布到若干散列桶中，一个值所属的散列桶由一个函数确定，该函数称为散列**(Hash)**函数，又称哈希索引（索引无序）。
- 搜索码（或称索引码）：用于文件中查找记录的属性或属性集称为搜索码**(search key)**，注意与前面介绍的“码”的区别，搜索码可以是任意一个属性而码则不可以。
- 索引记录：由搜索码和指向具有该搜索码值的指针组成



概念理解：顺序索引

- 顺序索引（**Sequential Index)**：将索引文件排序后保存，因而在索引文件中检索搜索码可以用二分法，计算复杂度为Log_2（N），当N ＞30时，就有显著效益。
- 按搜索码排序情况，有序索引分为两类：主索引（又称聚集索引），“小有序管大有序”；辅助索引（又称非聚集索引），“小有序管大无序”。
- 按索引密度，有序索引又分为两类：稠密索引；稀疏索引

概念理解：无序索引

- 从映射的角度理解无序索引
- 索引映射 **IndexMap**：{搜索码}→{记录号}，索引文件不排序。平均搜索次数为：桶搜索码总数/2 。
- 例：上述按属相分组后按组建索引，为无序索引，**30→30 mod 12→ 6组** → 组内找 **“30”**。组内可顺序查找，组小，加快了检索内容的速度。
- 典型的无序索引： **hash索引**

索引技术评价指标

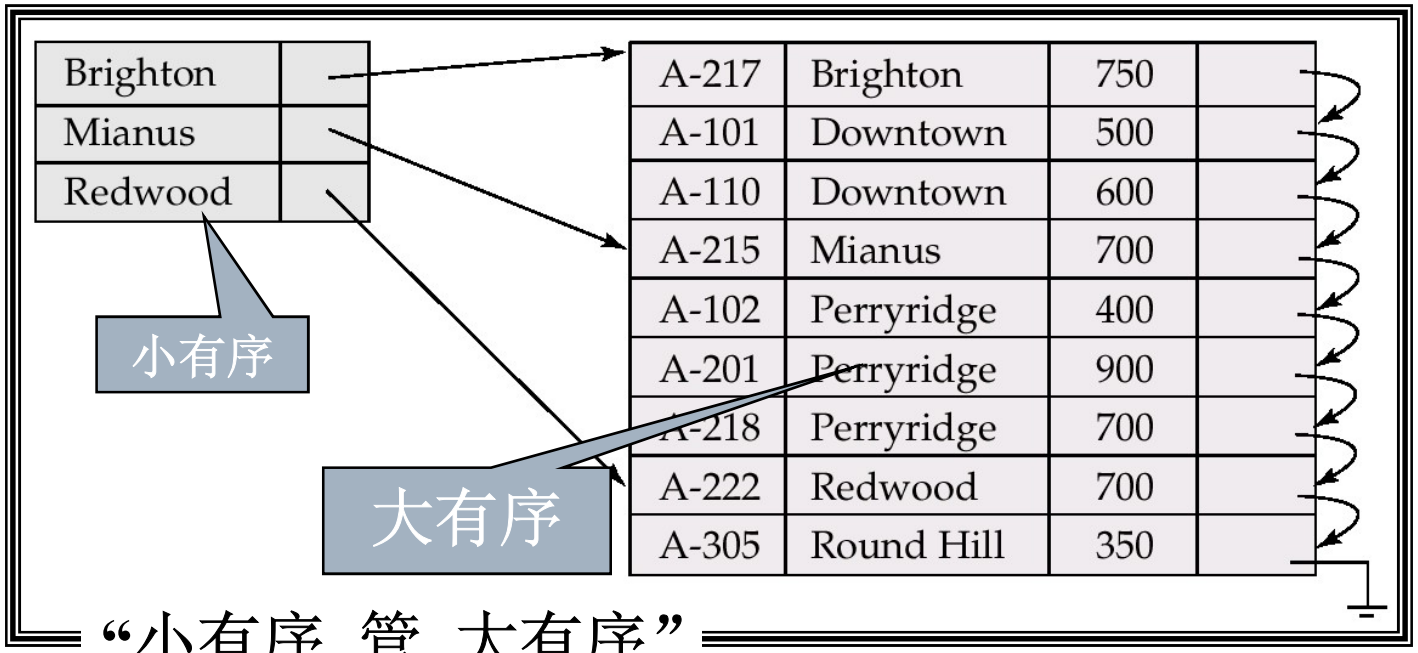
- 访问类型：能支持的访问类型。访问类型包括找到具有特定属性值的所有记录（等值索引），以及找到其属性值落在某个特定范围内的所有记录（比较索引）。
- 时间开销
  - 访问时间：用该技术找到一个特定数据项集所需要的时间
  - 插入时间：插入一个新的数据项所需时间，包括查找到插入位置及更新索引结构所需的时间。
  - 删除时间：删除一个数据项所需要的时间，包括找到待删除项所需时间和更新索引结构所需时间。
- 空间开销：一个索引结构所占用额外存储空间。

目录

- 问题的提出
- 数据存储
- 索引基本概念
- 有序索引
  - 主索引与辅助索引
  - 稠密索引与稀疏索引
  - 多级索引
  - 索引更新
  - B^+树索引
- 散列
  - 静态散列
  - 动态散列
  - 顺序索引与散列比较
  - 位图索引（自学）

主索引

- 主索引：如果包含记录的文件按照某个搜索码（对应表的属性）指定的顺序排序，那么该搜索码对应的索引称为主索引，又称聚集索引
- 索引文件远小于被索引的文件

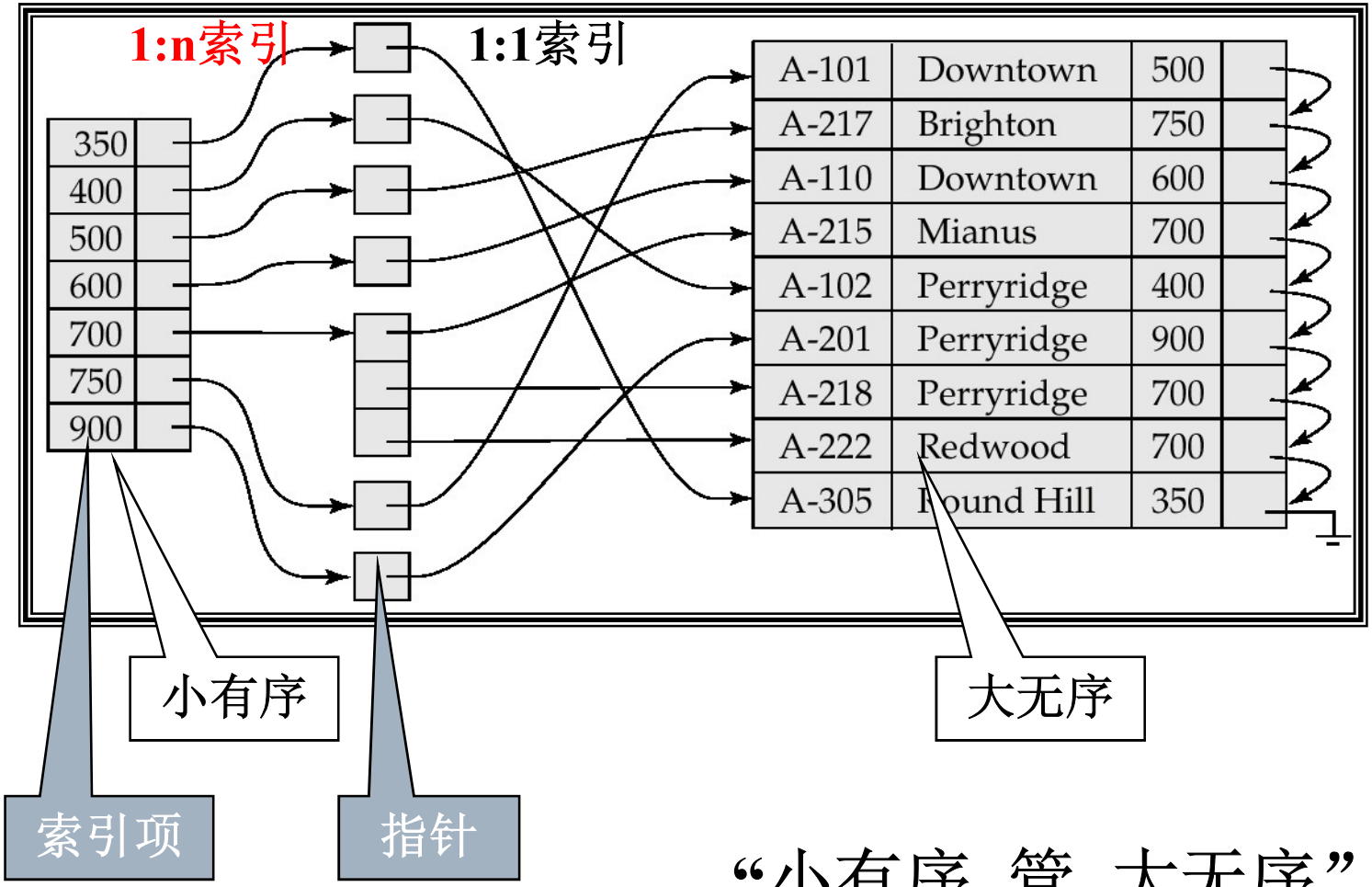


“小有序 管 大有序”

辅助索引

- 辅助索引：搜索码的在文件中无排序的索引称为非聚集索引，又称辅助索引。
- 即，记录的存储顺序与索引顺序不同的索引
- 辅助索引对每个搜索码值都有一个索引项，而且文件中的每个记录都有一个指针，因此辅助索引是稠密的。
- 辅助索引的结构：索引+指针桶 (附加的间接指针层)

account文件的辅助索引，搜索码：balance



“小有序 管 大无序”

比较：主索引与辅助索引

- 主索引＋顺序扫描，快（因按搜索码排序）
- 辅助索引＋顺序查找，慢（没有按搜索码排序，需要访问更多的块）
- 辅助索引能够提高主索引搜索码以外的搜索码的查询性能，但是，辅助索引增加了数据更新的开销。需要根据查询的频率来确定哪些辅助索引是需要的。
- 辅助索引必须是密索引，主索引可以是密索引也可以是稀索引

目录

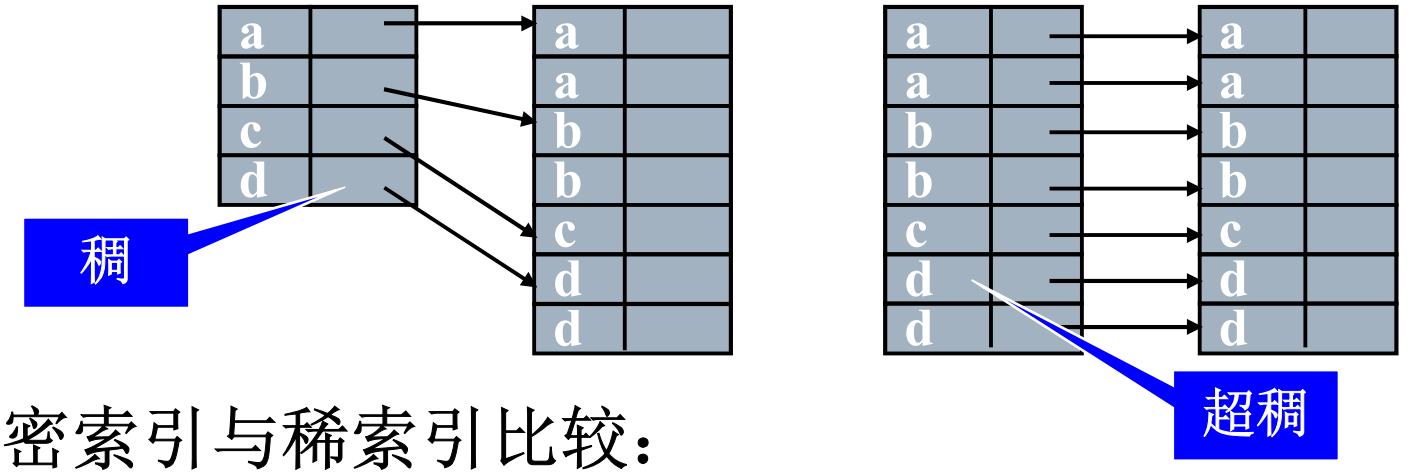
- 问题的提出
- 数据存储
- 索引基本概念
- 有序索引
  - 主索引与辅助索引
  - 稠密索引与稀疏索引
  - 多级索引
  - 索引更新
  - B^+树索引
- 散列
  - 静态散列
  - 动态散列
  - 顺序索引与散列比较
  - 位图索引（自学）

稠密索引与稀疏索引

- 根据索引密度，有序索引可以分为两类：
  - 稠密索引：文件中每个搜索码都有一个索引记录。
    - 具有相同搜索码值的其余记录顺序地存储在第一个数据记录之后。（稠）
    - 也可能存储指向具有相同搜索码的全部记录的指针。（超稠）
  - 稀疏索引：只有部分搜索码值建立了索引记录。

密索引

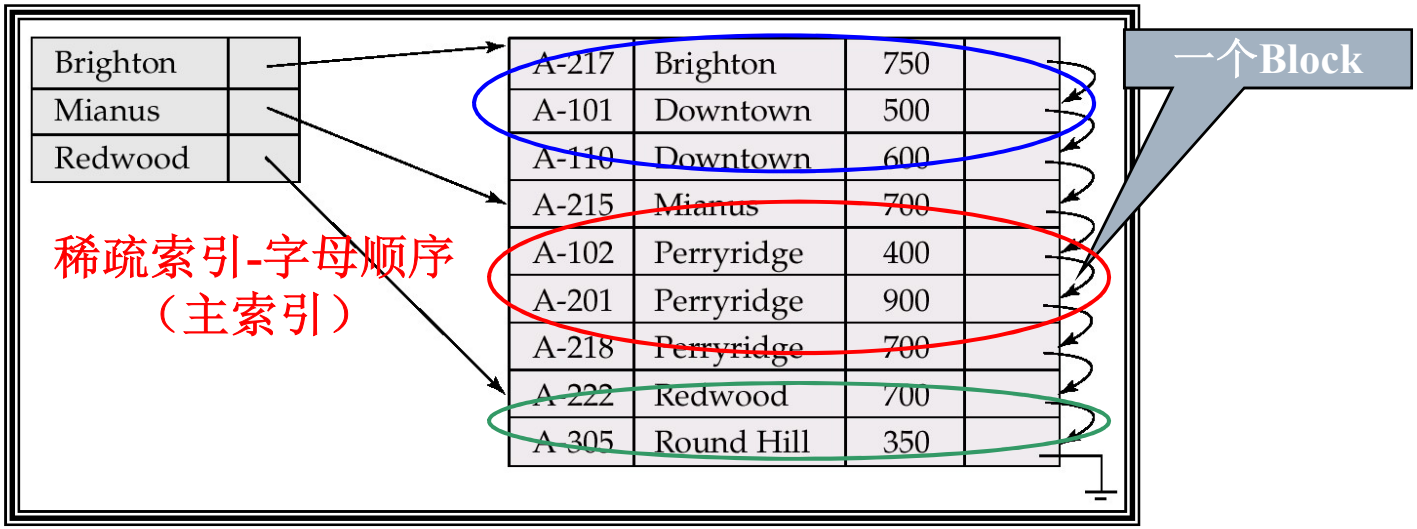
- 索引项 与 记录1-1 对应，一Key索一记录。 搜索码（无重复）1-1，稠，搜索码（有重复）1-1，超稠！



- 密索引与稀索引比较：
  - 稠密索引可以更快地定位一条记录，但稀疏索引占用的空间更少。采取哪种形式更好需要在时间和空间开销上进行权衡。

稀索引

- 设，一Key索一块，Key →Block块中顺序搜索，块可全装入内存。
- 新华字典：页眉索引，1-1页（一Key索一页），
- 优点：索引文件小，可全入内存
- **slower than dense index for locating records.**比密索引慢一点



目录

- 问题的提出
- 数据存储
- 索引基本概念
- 有序索引
  - 主索引与辅助索引
  - 稠密索引与稀疏索引
  - 多级索引
  - 索引更新
  - B^+树索引
- 散列
  - 静态散列
  - 动态散列
  - 顺序索引与散列比较
  - 位图索引（自学）

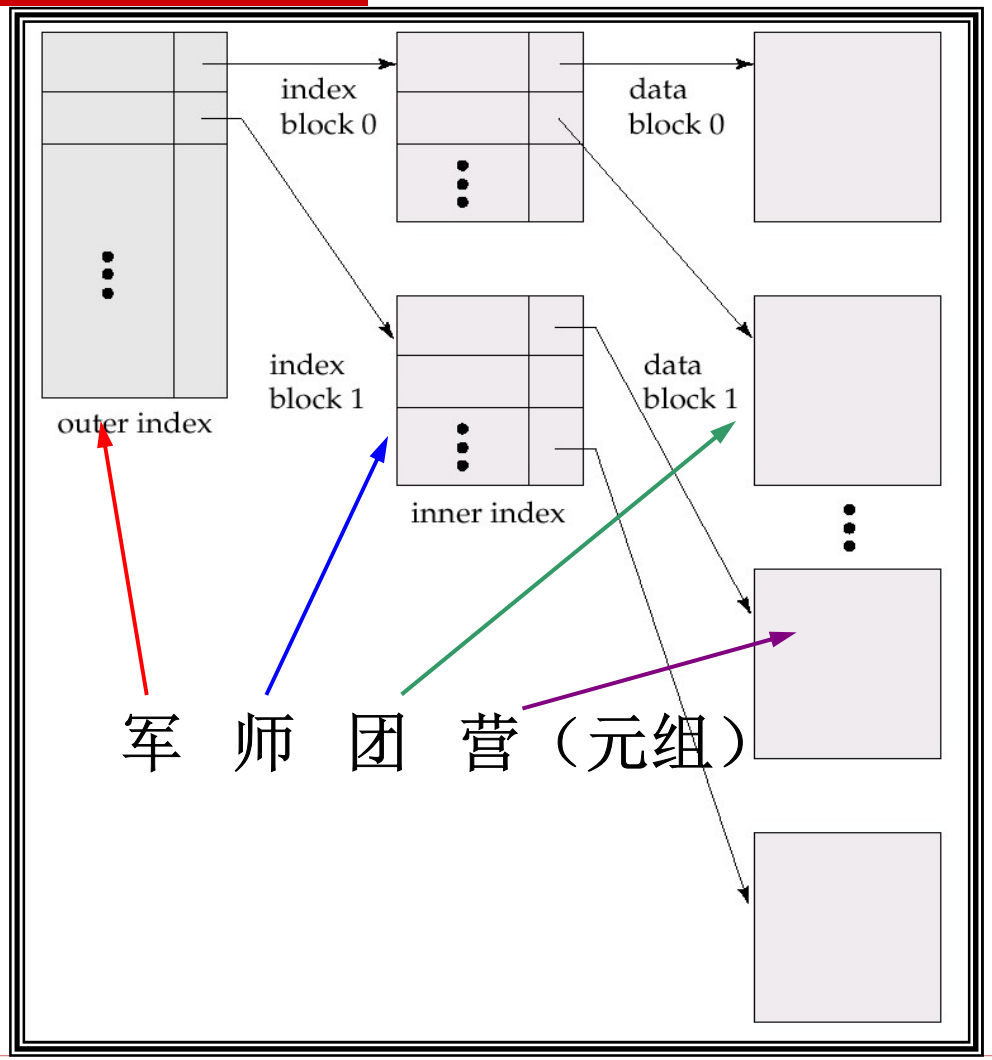
多级索引-问题及解决方案

- 问题：如果被索引的文件很大，即使采用稀疏索引也会变得很大，那么搜索索引系统开销将会很大，如何解决这一问题？
- 解决方法有两种：
  - 采用特殊技术提高搜索效率，如二分搜索定位搜索项，即如果索引占用**b**个块，需要读取$\log_2[b]$个磁盘块。这种方法对于大文件仍然需要很多时间。
  - 对索引文件进一步建立索引，即采用多级索引的方式解决。

多级索引的基本思想

- 在索引之上再建索引（一般为稀疏索引），形成外层索引和内层索引，在外层索引之上还可以进一步建立外层索引。
- 定位记录的方法：首先在外层索引上使用二分搜索找到其最大搜索码值小于或等于所需搜索码值的记录，指针指向内层索引块，直到找到其最大搜索码值小于或等于所需搜索码值的记录，这一记录指针指向所要查找记录的文件块。

二级稀疏索引
(内层索引-索引块)



目录

- 问题的提出
- 数据存储
- 索引基本概念
- 有序索引
  - 主索引与辅助索引
  - 稠密索引与稀疏索引
  - 多级索引
  - 索引更新
  - B^+树索引
- 散列
  - 静态散列
  - 动态散列
  - 顺序索引与散列比较
  - 位图索引（自学）

索引更新

- 当文件中的记录进行删除或插入时，索引需要更新。
- 索引更新包括：
  - 单级索引更新：插入、删除
  - 多级索引更新：单级索引可以更新了，扩展、递推

稠密索引-插入

- 单级索引插入：
  - 稠密索引：
    - 插入元组的搜索码值不在索引中，插入该搜索码的索引记录。
    - 否则如果索引记录存储的是指向所有搜索码值的所有记录指针，系统在索引记录中增加一个指向新记录的指针。（超稠）
    - 否则如果索引记录存储一个仅指向具有相同搜索码值的第一个记录的指针，系统就把待插入的记录放到具有相同搜索码值的其他索引记录之后（稠）

- 稀疏索引：
  - 假设索引为每个块保存一个索引项。如果系统产生一个新块，将新块中出现的第一个搜索码值插入到索引中。
  - 如果新插入的记录含有块中的最小搜索码，那么系统就更新指向该块的索引项(指针)
  - 否则系统对索引不做任何改动

单级索引-删除

- 系统首先找到待删除的记录
- 稠密索引：
  - 被删除的记录唯一具有这个特定搜索码值的记录，系统从索引中删除相应的索引记录
  - 否则如果索引记录存储的是指向所有相同搜索码值记录的指针，系统从索引记录中删除指向被删除记录的指针。（超稠）
  - 否则如果索引记录存储的是指向具有该搜索码值的第一条记录的指针，如果删除记录是第一条记录，系统更新索引记录并指向下一条记录（稠）
- 稀疏索引：
  - 如果索引中不包含被删除记录搜索码值的索引记录，则索引不做修改
  - 否则系统采取如下操作：

(1) 如果被删除记录是具有该搜索码值唯一记录，系统用下一个搜索码值的索引记录替换相应的索引记录。如果下一个搜索码值已经有了一个索引项，则删除而不是替换该索引项。

(2) 否则，如果该搜索码值的索引记录指向被删除的记录，系统就更新索引记录，使其指向具有搜索码值的下一条记录

- 多级索引的插入和删除算法是对上述算法的一个简单扩充，系统从底层索引逐层向上更新索引，更新方法同单级索引。

索引顺序文件的最大缺点—
文件增大导致索引增大
索引增大导致性能下降

目录

- 问题的提出
- 数据存储
- 索引基本概念
- 有序索引
  - 主索引与辅助索引
  - 稠密索引与稀疏索引
  - 多级索引
  - 索引更新
  - **B^+树索引**
- 散列
  - 静态散列
  - 动态散列
  - 顺序索引与散列比较
  - 位图索引（自学）

关于B树和B+树

- B树最早由Bayer和McCreight两人1972年提出的，后来在数据库的文件组织中得到广泛应用。B树与普通树的区别在于对树中各结点的孩子数目和每条路径的长度有一定的限制：

  (1) 根结点至少有2个孩子(除非它本身又是叶子)

  (2) 树中所有叶子都在同一级上(平衡树)；

  (3) 除根、叶外，每个结点不少于d个孩子；

  (4) 每个结点不多于2d－1个孩子。

  上述树称为d阶B树。

- 以索引块为结点并允许存在存储冗余的B树为B^+树，B^+树是B树的一个变种。参阅数据结构。

B+树索引

- B+树索引文件是有序索引
- B+树索引采用平衡树结构，树根到叶子的每条路径长度相同
- B+树索引是多级索引，但结构不同于上述多级索引顺序文件

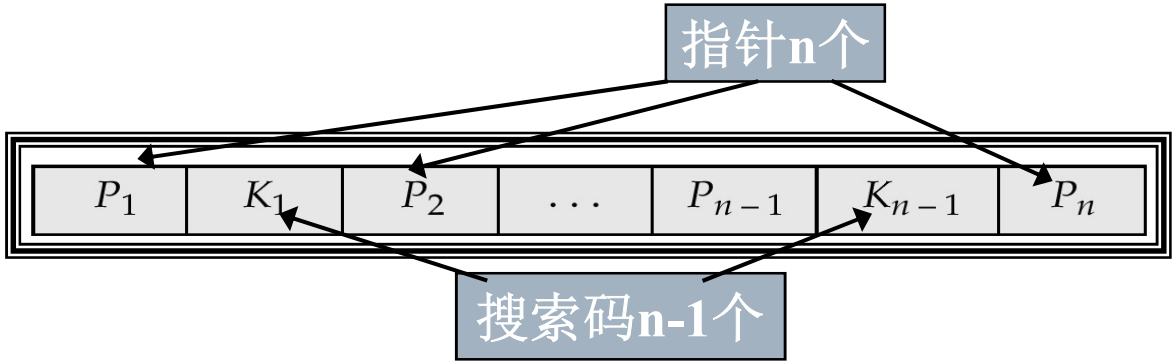
- B+树索引的时间开销和空间开销对于更新频率高的文件来说是可以接受的
- B+树索引是使用最广泛的在数据插入和删除情况下仍能保持其执行效率的一种索引结构

B^+树的结构

- 结点结构：
  - 最多包含n-1个搜索码值$K_1, K_2, \ldots, K_{n-1}$，以及n个指针$P_1, P_2, \ldots, P_n$。每个结点中的搜索码值排序存放，如果$i<j$，那么$K_i < K_j$
  - 对于$i=1, 2, \ldots, n-1$, 指针P_i指向具有搜索码K_i的一个文件记录或指向一个指针桶，而桶中的每个指针指向具有搜索码值K_i的一个文件记录
  - 叶结点的指针指向文件记录
- B^+树的非叶结点构成叶结点上的一个多级(稀疏)索引
- 叶结点和非叶结点结构相同

B+树结点结构

- 典型的B+树结点结构



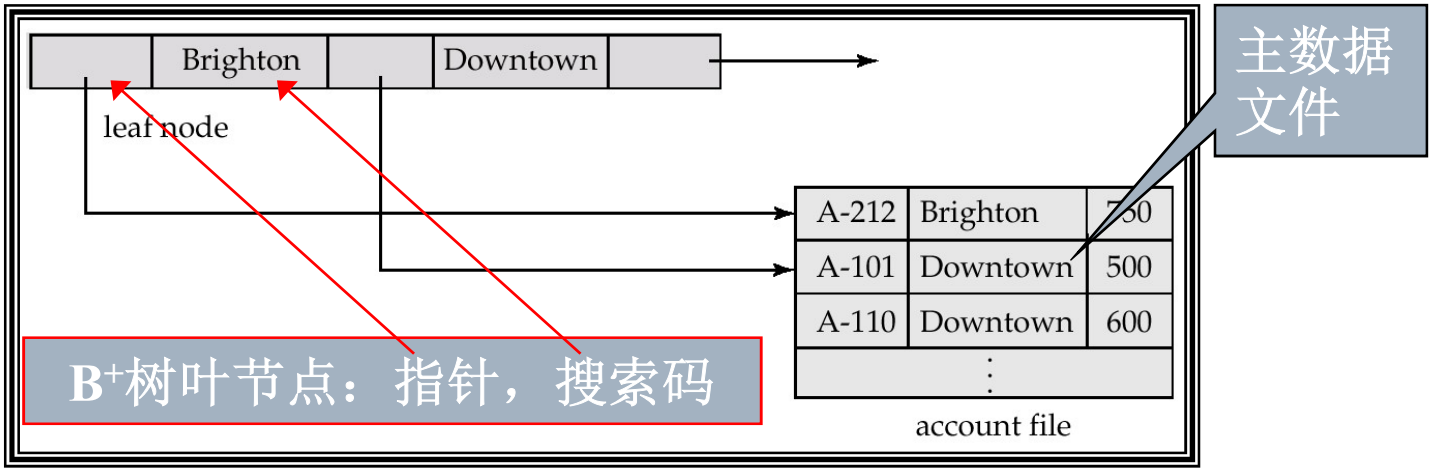
K_i are the search-key values
P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).

The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$ 搜索码排序了

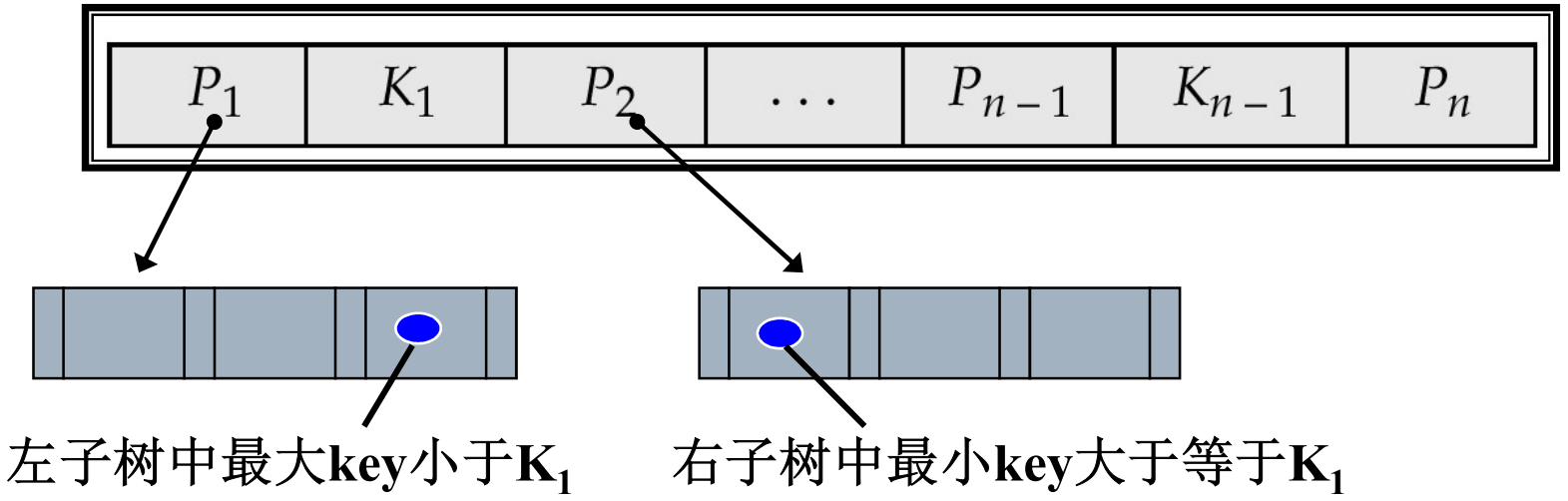
叶节点

- **account的B+树索引的一个叶结点 (n=3, 结点指针数量)**
- For i = 1, 2, . . ., n–1, pointer P_i either points to a file record with search-key value K_i, or to a bucket of pointers to file records, each record having search-key value K_i. Only need bucket structure if search-key does not form a primary key.
- 允许叶节点最少有**[(n-1)/2]**个值
- P_n points to next leaf node in search-key order



非叶节点

- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with m pointers:
  - For $2 \le i \le n-1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_{m-1}　　半（每个节点至少一半有孩子）
  - All the search-keys in the subtree to which P_1 points are less than K_1　序

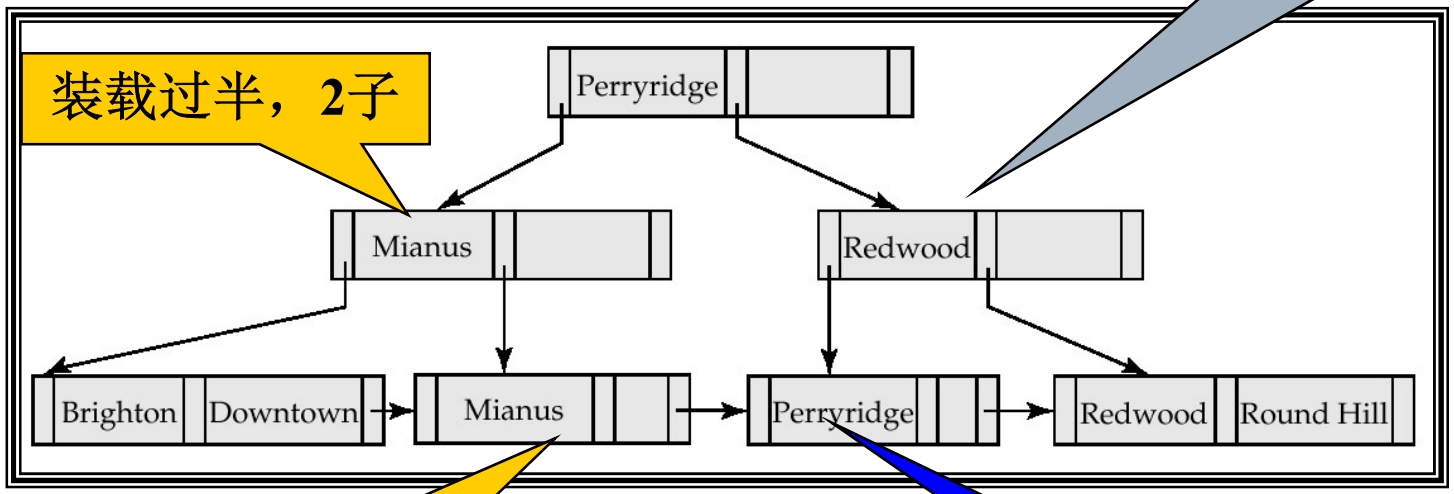


B+树的优缺点

- B+树的优点：
  - 平、半、序（原则）、矮、胖（特点）
  - 在生长中保持平衡（叶根距 一致 ），装载因子过半（**Each node that is not a root or a leaf has between [*n*/2] and *n* children**，中间节点指针装载要过半，根（不算）、叶（无子，值装载要半**[(n-1)/2]**），**n**为指针数量，又称扇出，**[*n*/2]**取整数部分），有序（搜索码有序）。
  - 自动重组，局部更新与时俱进
- B+树的缺点：结点分裂合并时的开销大
  - 插入删除时可能递归作几次，装载因子**50—75%**
  - 利大于弊，广泛采用

例子： B^+树

account文件的完整的B^+树(n=3)(注意P_n)



非叶结点中key为分水岭，左小右大

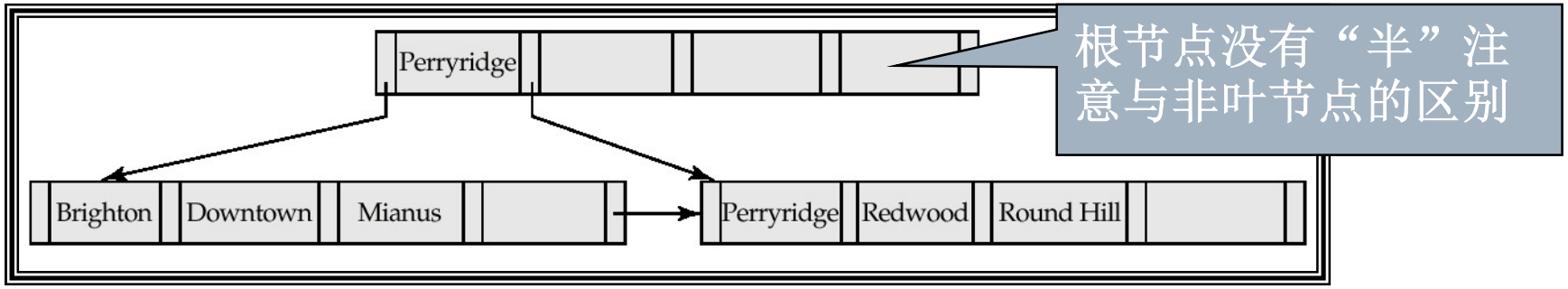
装载过半，2子

装载过半，(3-1)/2

近距离顺序查找，沿着叶子指针找，搜索码是支行名称，按字典顺序

扁平的B+树

account文件的完整的B+树(n=5)



- 叶结点上项满4半2（至少一半[(5-1)/2]）
- **Root must have at least 2 children.** 根至少有2孩子
- 为提高查询效率，实际系统中的B+树多为扁平的，矮而胖。

B+树的查询

- 查询搜索码值V。首先检测根结点，找到大于V的最小搜索码值，如果找到了这个搜索码值是K_i，然后顺着指针P_i到达另一个结点。如果找不到这样的值，则$k \geq K_{m-1}$，其中m是该结点中的指针数，则沿着P_m到达另一个结点。
- 以此类推，最终到达一个叶结点。如果在该叶结点中有某个搜索码值$K_i=V$，那么指针P_i指向所需要的记录。如果该叶结点中找不到值V，则不存在码值为V的记录。
- 处理一个查询需要遍历树中从根到叶结点的一条路径，如果文件有K个搜索码值，那么这条路径的长度不超过$|\log_{|n/2|}(K)|$个磁盘块（代价），非准确值。

例子：B+树搜索效率分析

设一个文件有100 0000条记录，一个磁盘块存储10条记录。采用对比分析法。

- 建立一个一级稀疏索引，需要10 0000条索引记录。
- 假设一个磁盘块能容纳100条索引记录，则共需要1000个磁盘块。
- 索引搜索均采用二分搜索。
- 一次搜索访问的磁盘块数量：
  - $b=\log_2 1000=9.97\rightarrow 10$(向上取整)
- 若每个磁盘块读操作为30ms，则搜索所用的时间为300ms

采用B^+树搜索：假设每个结点占用一个磁盘块，设为4KB。搜索码大小为32字节，指针为8字节，则n≈100（4KB=(n-1)×(32+8)+8，每个节点的指针数，节点扇出，n=104）。则一次搜索需要访问的结点数量为:

$$|\log_{|n/2|}(K)| = |\log_{52}(1000000)| = 3.5 \rightarrow 4(\text{磁盘块})$$

所需时间不超过120ms（4×30ms），小于300ms。注意，K取100 0000，最大可能值（无重复搜索码值），即最坏情况。

一般B^+树搜索所需访问的磁盘块为3个或更少，等于树的高度，因此，B^+树多为扁平的，矮而胖.

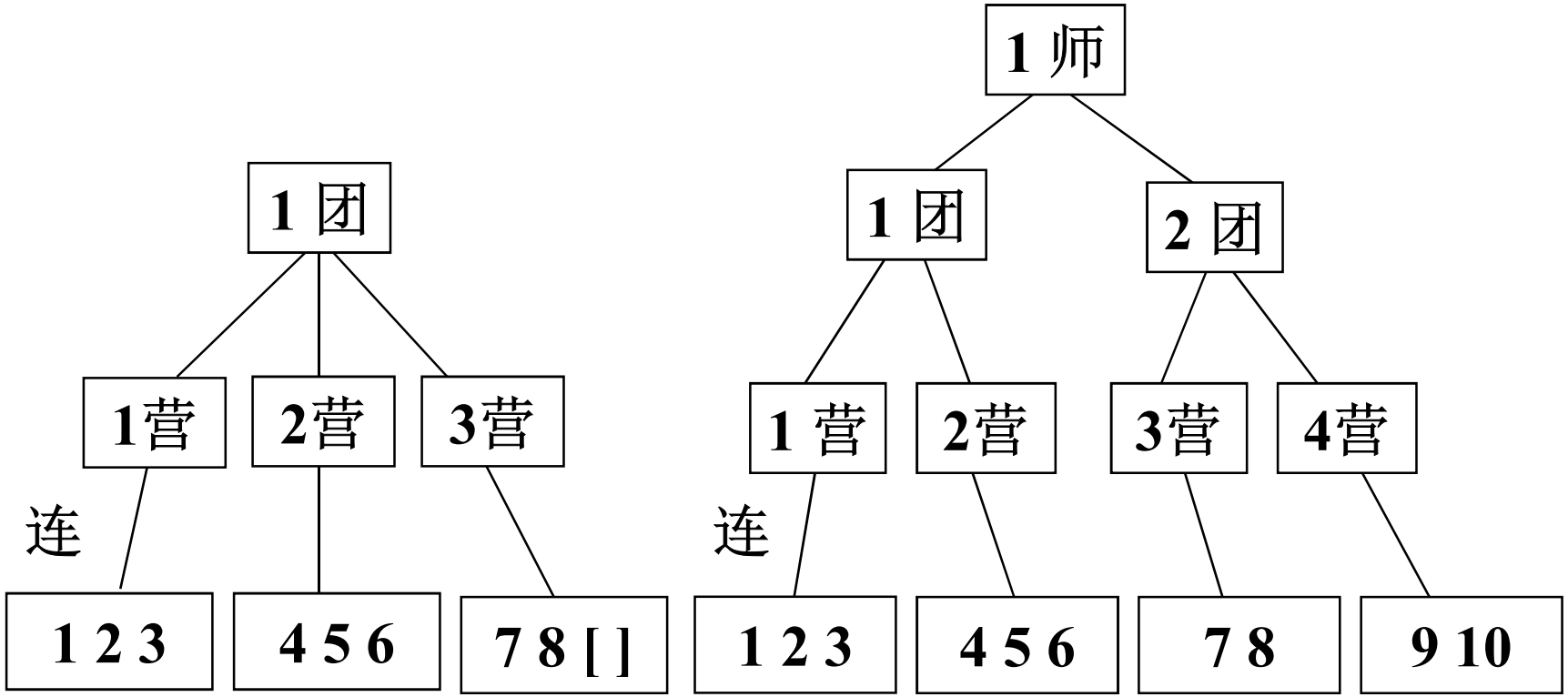
B^+树的更新-插入-分裂

- B^+树在插入时可能会变得过大而需要分裂, 在删除时可能会因为过小(指针数小于[n/2])而需要合并。
- 插入→分裂，删除→合并
- 插入-分裂：分裂一个结点后，将新的叶结点插入到B^+树结构中，若父结点有空间则插入，否则父结点也要进行分裂，以此类推。如果根结点也分裂则整棵树的深度就增加。
- 删除-合并：有两种情况，（1）无合并：删除一个叶结点，必须从其父结点删除指向它的指针。（2）合并：父结点被删除后, 兄弟结点合并
- 分裂合并的原则：平、半、序

B+树的更新 理解：部队编制

- 用部队编制常识来说明 B+树在插入或删除时，节点的分裂与合并。以及B树高度的增减和根节点浮动等递归过程。
- 部队的编制，叶节点为连队，三三制：
  - 一个师至多3个团（n=3），至少2个团（半）
  - 同样，一 个团至多3个营，至少2个营，一个营应至多3个连，至少2个连。
  - 用[ ] 表示缺编的空白位置。
- 例子：有一个团，如 (a)，根节点为团级，叶节点为连级。
  - 以部队番号（即层次序列码）为搜索码，例如，1团2营4连（理解为字符串）

例子：部队编制

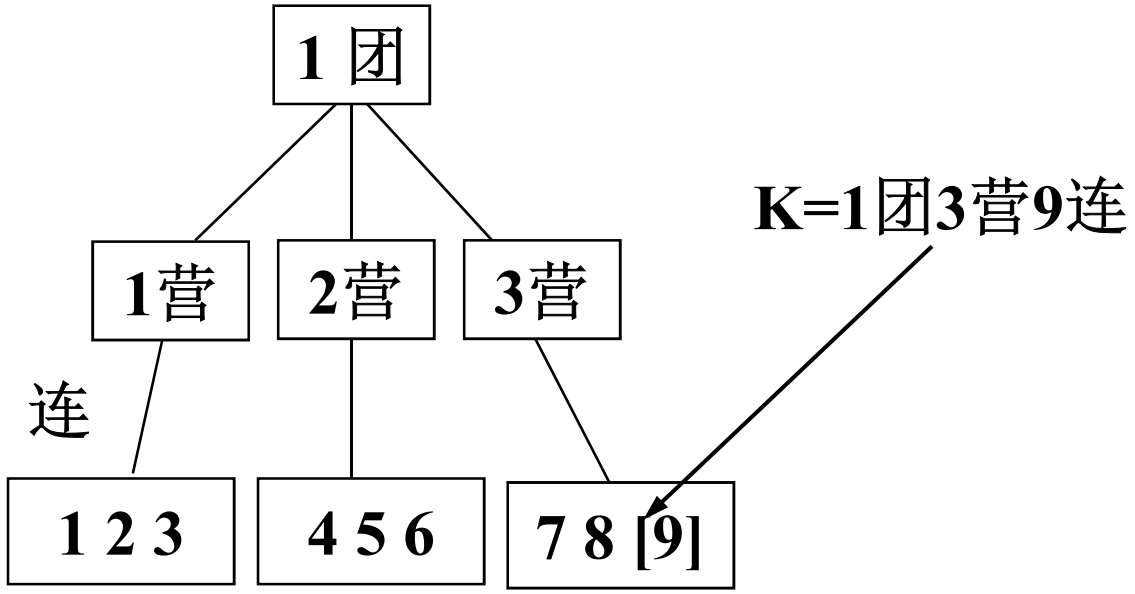


（a）　　（b）

增加第9 、10连后一个编制树，后面解释

例子：部队编制-插入不分裂

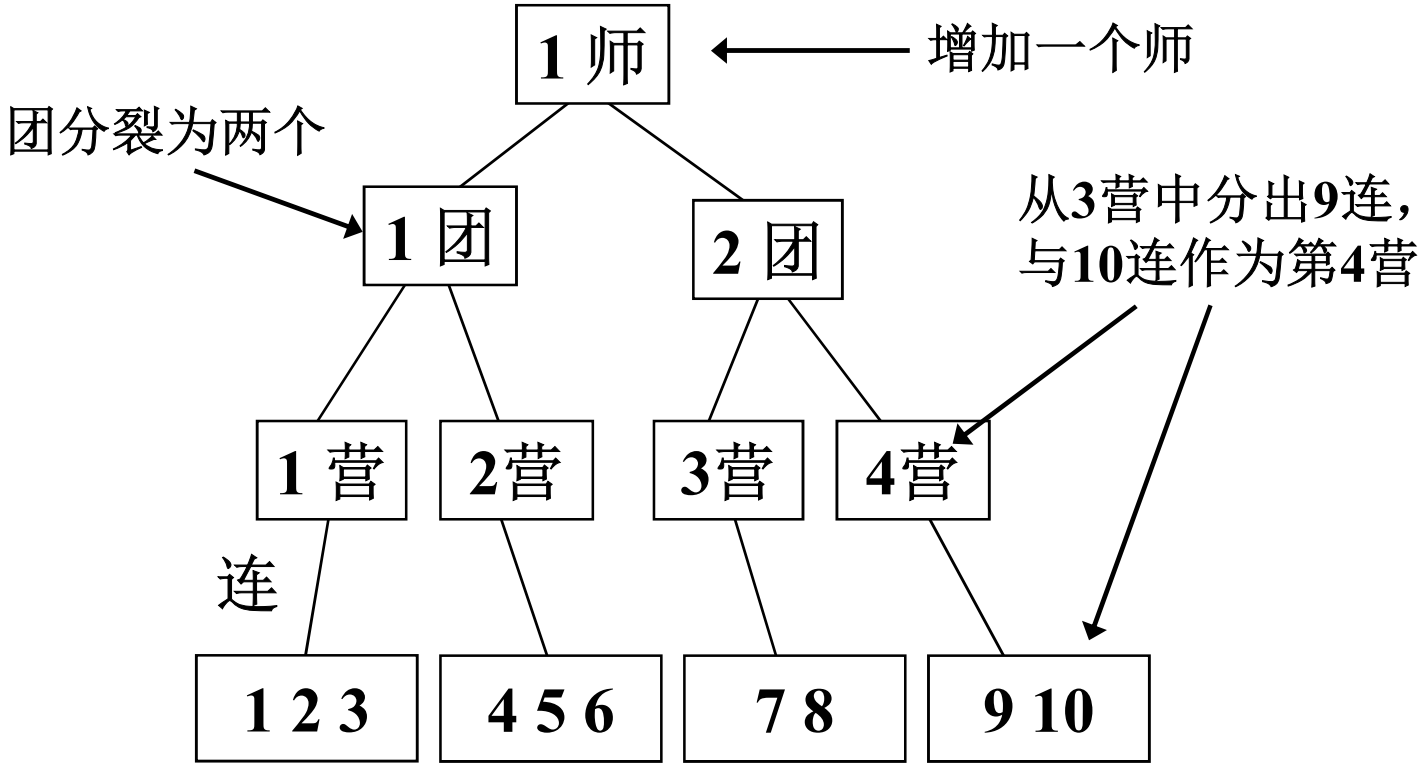
- 不发生节点分裂的插入：
  - 插入搜索码K=1团3营9连的索引项。
  - 先搜索K所在项，得到其位置。
  - 在[ ]处，现为空白，填入K=9即可。



例子：部队编制-插入分裂

- 发生节点分裂的插入：
  - 再插入K=1团3营10连的索引项。
  - 按搜索码搜索，似应插入到1团3营9连右方。
  - 但如此则3营有4个连，超编
- 分裂节点如下：
  - 从3营中分出9连，与10连作为第4营，保半、序。
  - 问题递归到上一层次，即增加第4营。
  - 于是1团中有4个营，超编。团分裂两个团：1 团、2 团新设一个师作为根节点。

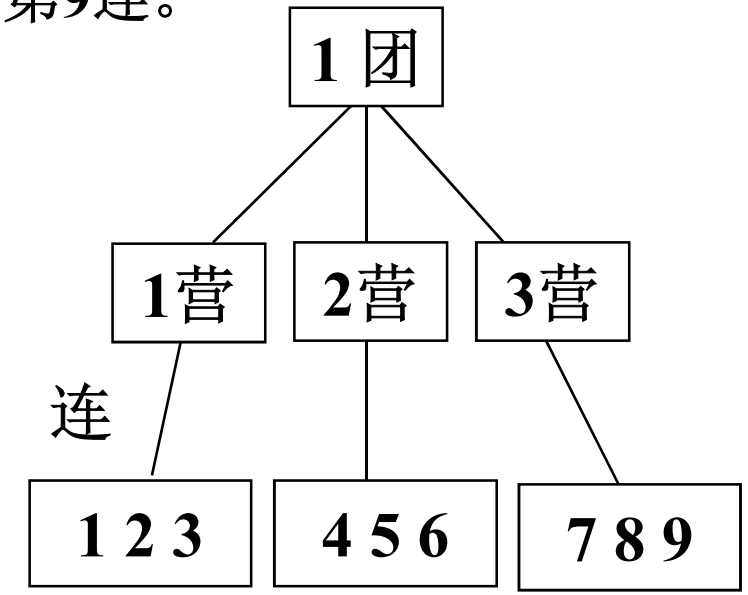
分裂结果



此师战斗实力甚虚☺

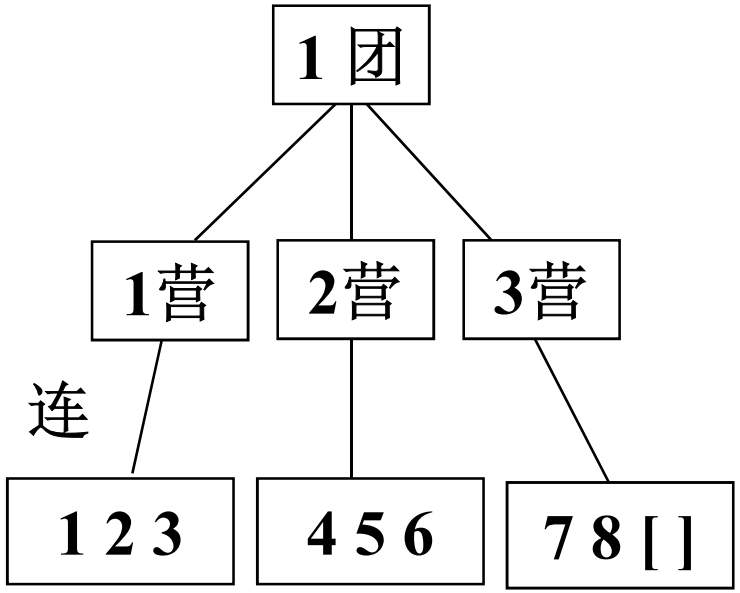
例子：部队编制-删除合并

- 删除合并情况：
  - 当从(b)中删除10连后，4营缺编。
  - 于是，4营与3营合并，导致2团缺编，2团与1团合并。
  - 于是，1师缺编，从而取消1师编制。
  - 结果相当于在图(a)中增加了第9连。



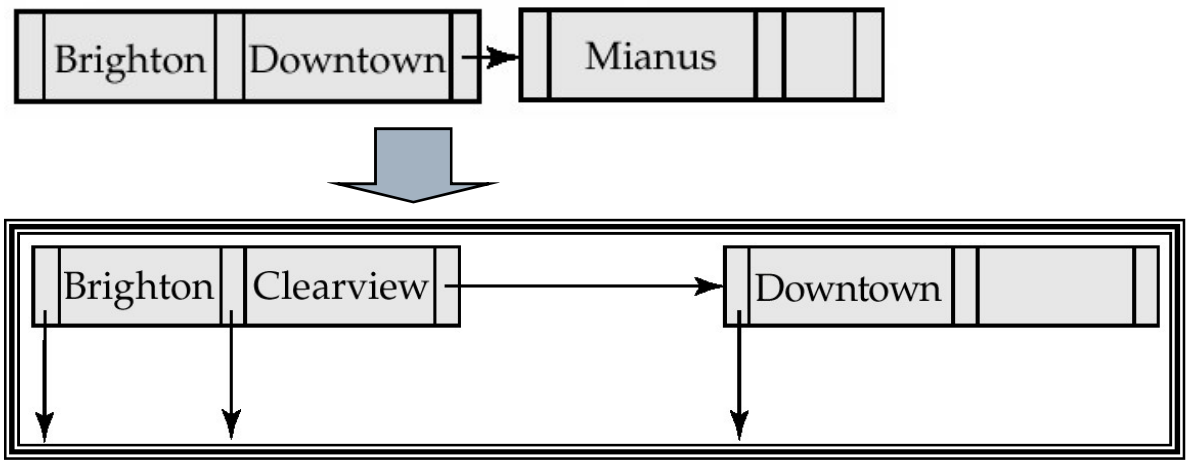
例子：部队编制-删除不合并

- 删除不合并情况：
  - 在上面结果中再删除1团3营9连，搜索到位后，改9连为空白。结果如图(a)所示。
  - 以上递归过程的 思路和大方向。

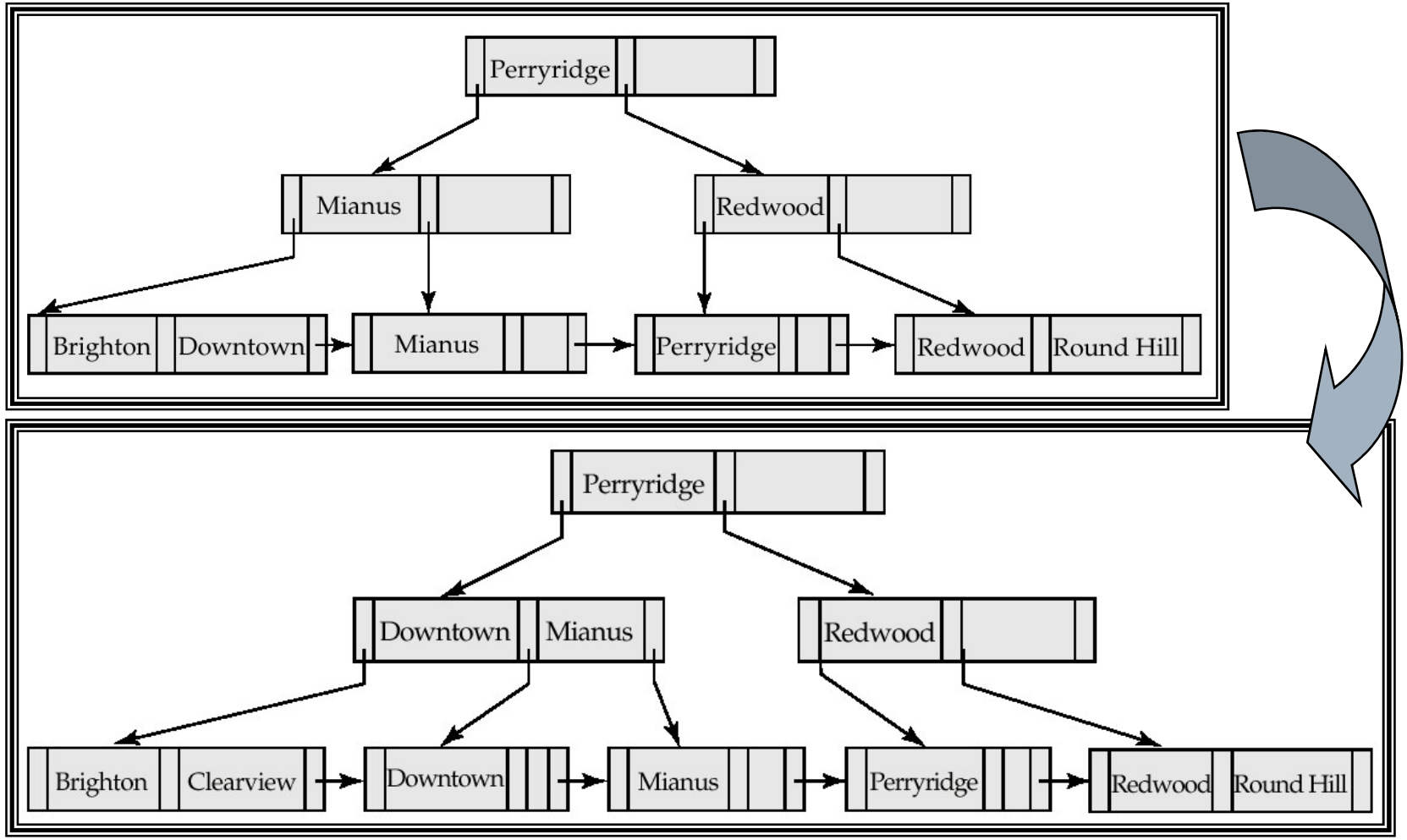


例子：B+树的更新-插入-分裂

- 结点分裂：如需要插入Clearview记录, 按照排序应该在叶结点Brighton/Downtown中,但该结点已经没有空间(假设 n=3, 则[3/2]=2), 这时需要将该结点分裂为两个结点。
- 由于Downtown是新叶结点的最小搜索码值,将该搜索码值插入到被分裂叶结点的父结点中,

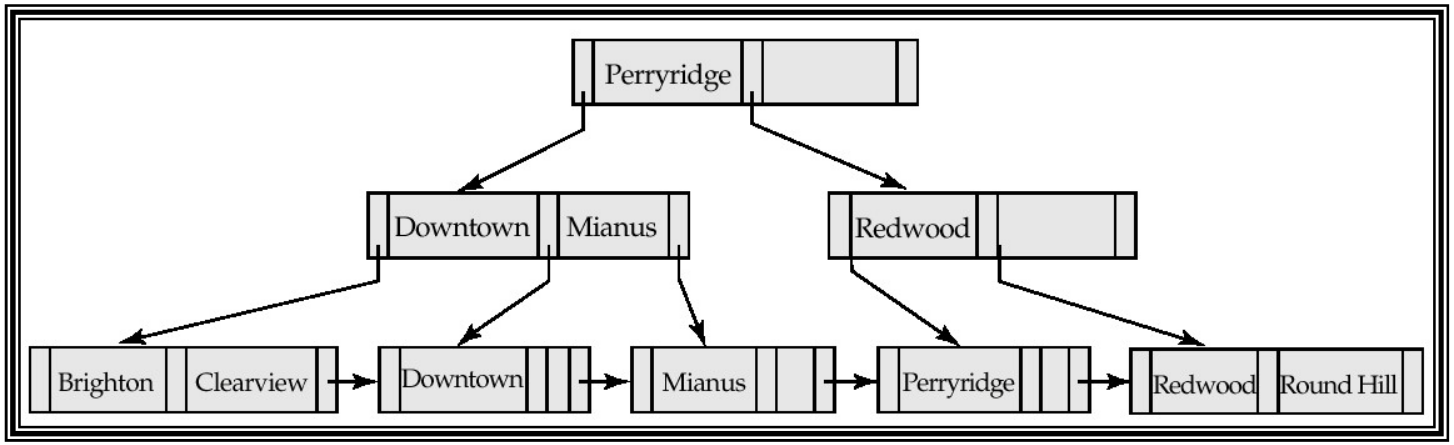


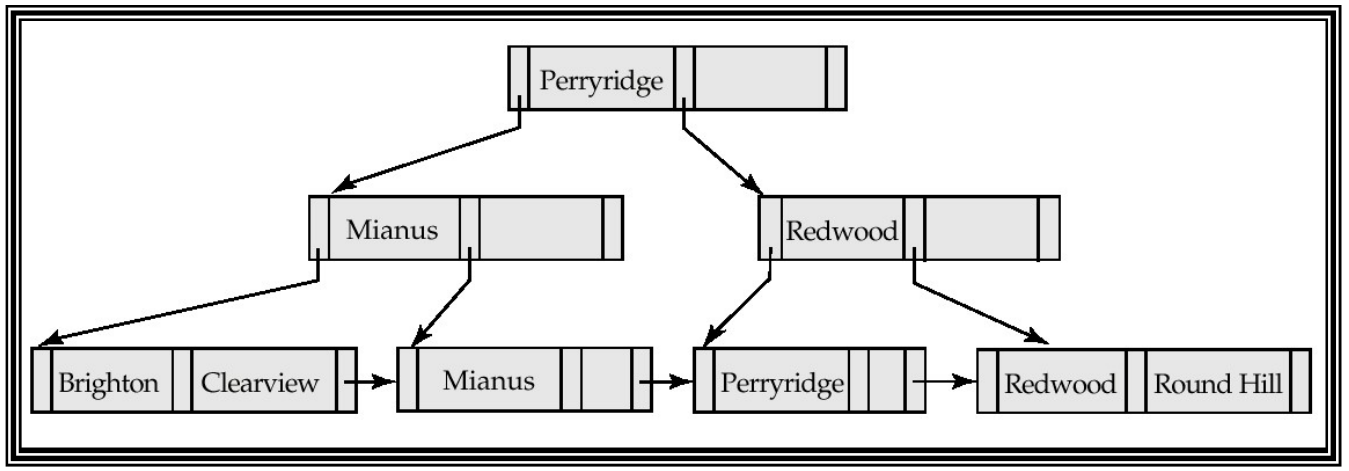
分裂后的完整的B^+树


Perryridge
Mianus
Redwood
Brighton
Downtown
Mianus
Perryridge
Redwood
Round Hill
Perryridge
Downtown
Mianus
Redwood
Brighton
Clearview
Downtown
Mianus
Perryridge
Redwood
Round Hill

B+树的更新-删除-合并

无合并：删除一个叶结点,必须从其父结点删除指向它的指针。

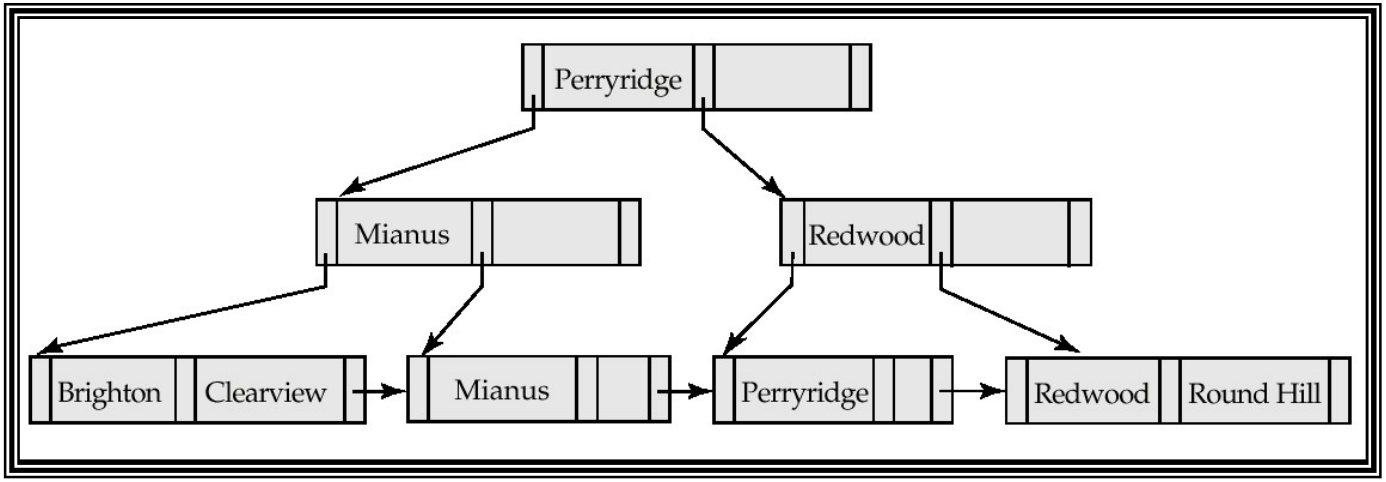


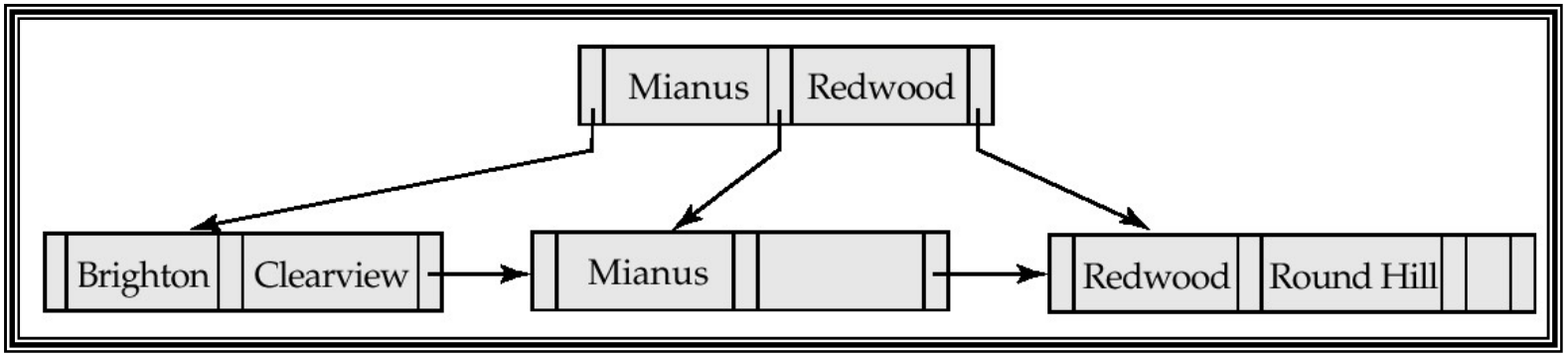


叶结点 **Downtown** 被删除

合并：

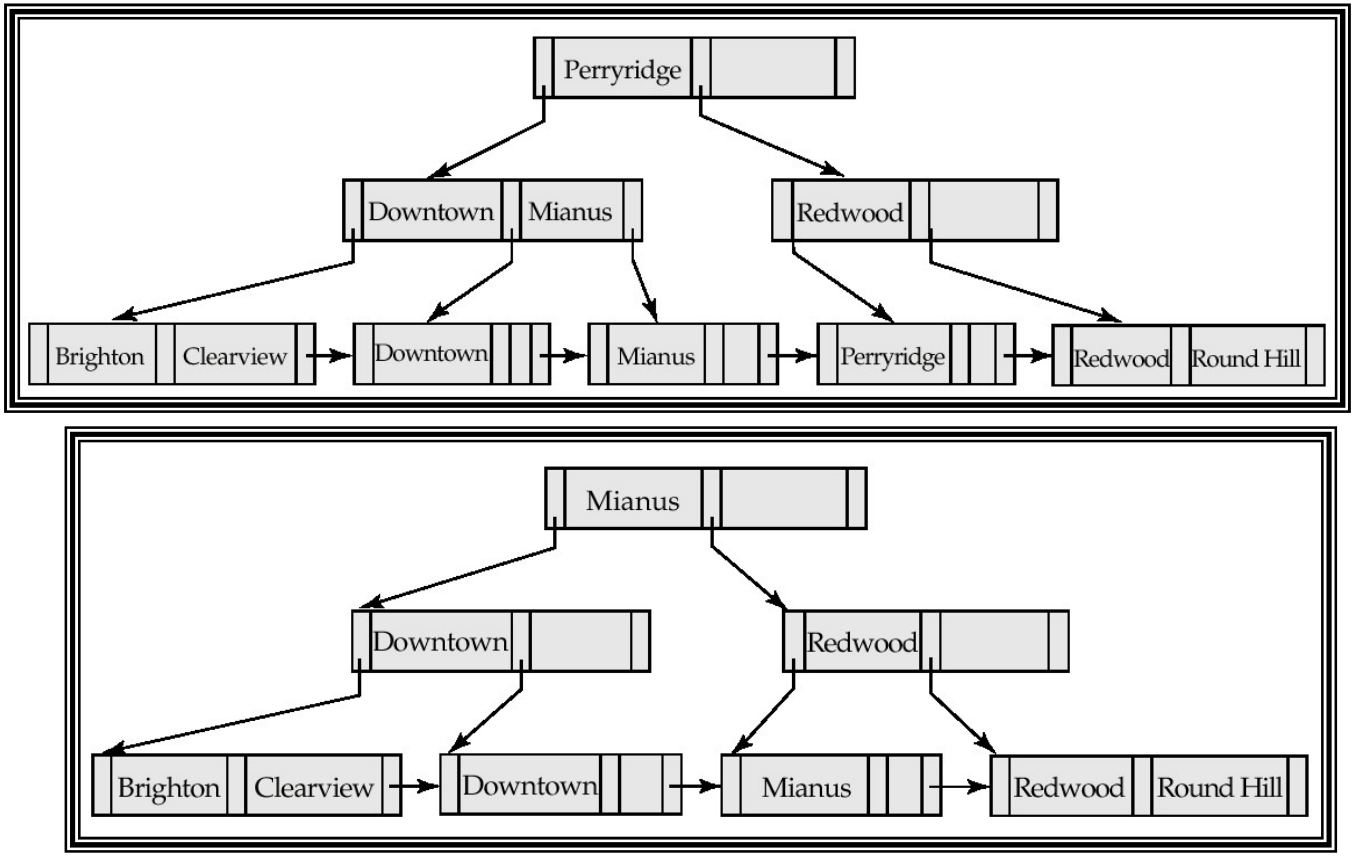
父结点(Perryridge)被删除后, 兄弟结点Mianus和Redwood合并





重新分布(重组)：

父结点 (**Perryridge**) 被删除后, 由于兄弟结点(**Downtown,Mianus**) 和**Redwood**无法合并，重组 - 将**Mianus**作为根结点后**Downtown**和**Redwood**做合并



B+树更新的优点

- 虽然更新过程复杂，但需要较少的I/O操作(I/O操作的代价很高)，所需代价正比于树的高度,一般B+树矮而胖，因此，效率很高，是数据库系统常用的索引结构

- B+树适合组织大型的索引文件。很多DBMS产品都采用B+树索引。

小结

- 索引是冗余的数据，对文件数据的正确性并不是必须的
- 由于索引结构提供了定位和存取数据的一条路径，因此，索引结构也称为存取路径
- 索引顺序文件—主索引，辅助索引，多级索引，在小数据量的文件中适用，大数据量的文件中效率会明显降低
- **B**$^+$树一定程度上克服了索引顺序文件存在的问题，适合大型数据索引。**B**$^+$树查找简单，但更新很复杂

目录

- 问题的提出
- 数据存储
- 索引基本概念
- 有序索引
  - 主索引与辅助索引
  - 稠密索引与稀疏索引
  - 多级索引
  - 索引更新
  - B^+树索引
- 散列
  - 静态散列
  - 动态散列
  - 顺序索引与散列比较
  - 位图索引（自学）

问题

- B^+树索引改善了顺序索引存在的问题：文件大→索引大→速度慢，但没有从根本上改变
- 通过索引结构定位数据，当文件过大时，系统开销也必然随之增大
- 有没有更好的办法—即数据定位与文件大小无关？

散列

- 散列的基本思想：通过一个函数将所有搜索码值均匀放在大小相同的桶中，要对某个搜索码进行访问只要用这个函数计算一下就可以找到相应的桶。
- 散列(Hash)函数：
  - 设K是所有查找搜索码值的集合，B是所有桶地址的集合。散列函数h(hash function)是一个从K到B的一个函数。

    K(搜索码) → Hash → B(桶地址)
  - 如，要访问记录Perryfidge，首先计算h(Perryfidge)=桶5的地址，找到桶5，在桶5内搜索Perryfidge即可。

散列的两个条件

- 使用散列方法，首先要有一个好的散列函数。由于在设计散列函数时不可能精确知道要存储的记录的查找搜索码值，因此要求散列函数在把查找搜索码值转换成存储地址（桶号）时，应满足下面两个条件：
  - 分布是均匀的：散列函数为每个桶分配所有可能的搜索码值集合中的同样数量的搜索码值。即每个桶的容量均匀。
  - 分布是随机的：所有散列函数值不受搜索码值顺序的影响，例如字母顺序、长度顺序等。

散列函数设计

- 一个糟糕的散列函数可能导致查找所花费的时间与文件中的搜索码数目成正比
- 一个设计良好的散列函数一般情况下查找所花费的时间是一个(较小的)常数，而与文件中搜索码的个数无关
- 尽管如此，不同的搜索码可能会有相同的散列函数值。即有可能：

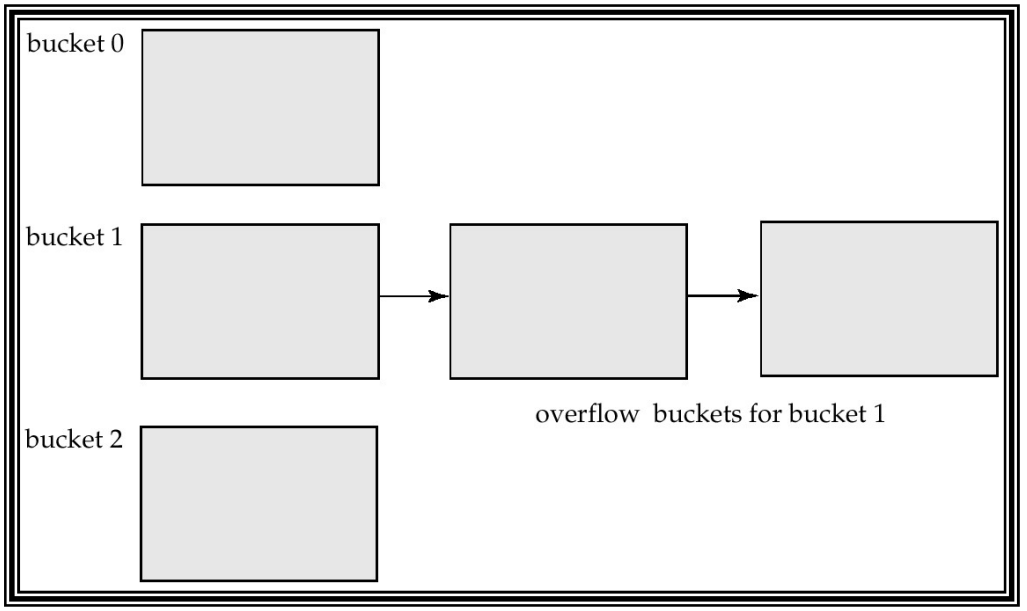
  $h(K_5)= h(K_7)$， K_5和K_7要放在同一个桶中
- 因此，当有多个这样K值时，可能会出现桶溢出现象。

桶溢出处理

- 如果一个桶没有足够的空间，就会出现桶溢出，桶溢出发生的原因：
  - 桶不足：桶数目n_B的选择必须保证$n_B > n_r / f_r$，其中n_r表示将要存储的记录总数，f_r表示一个桶中能存放的记录数目
  - 偏斜：桶足，但某些桶分配的记录比其他桶多，即使其他桶仍然有空间，某些桶仍然有可能溢出。(原因：多个记录可能具有相同的搜索码；所选的散列函数可能会造成搜索码分布不均匀)

桶溢出的处理–溢出桶

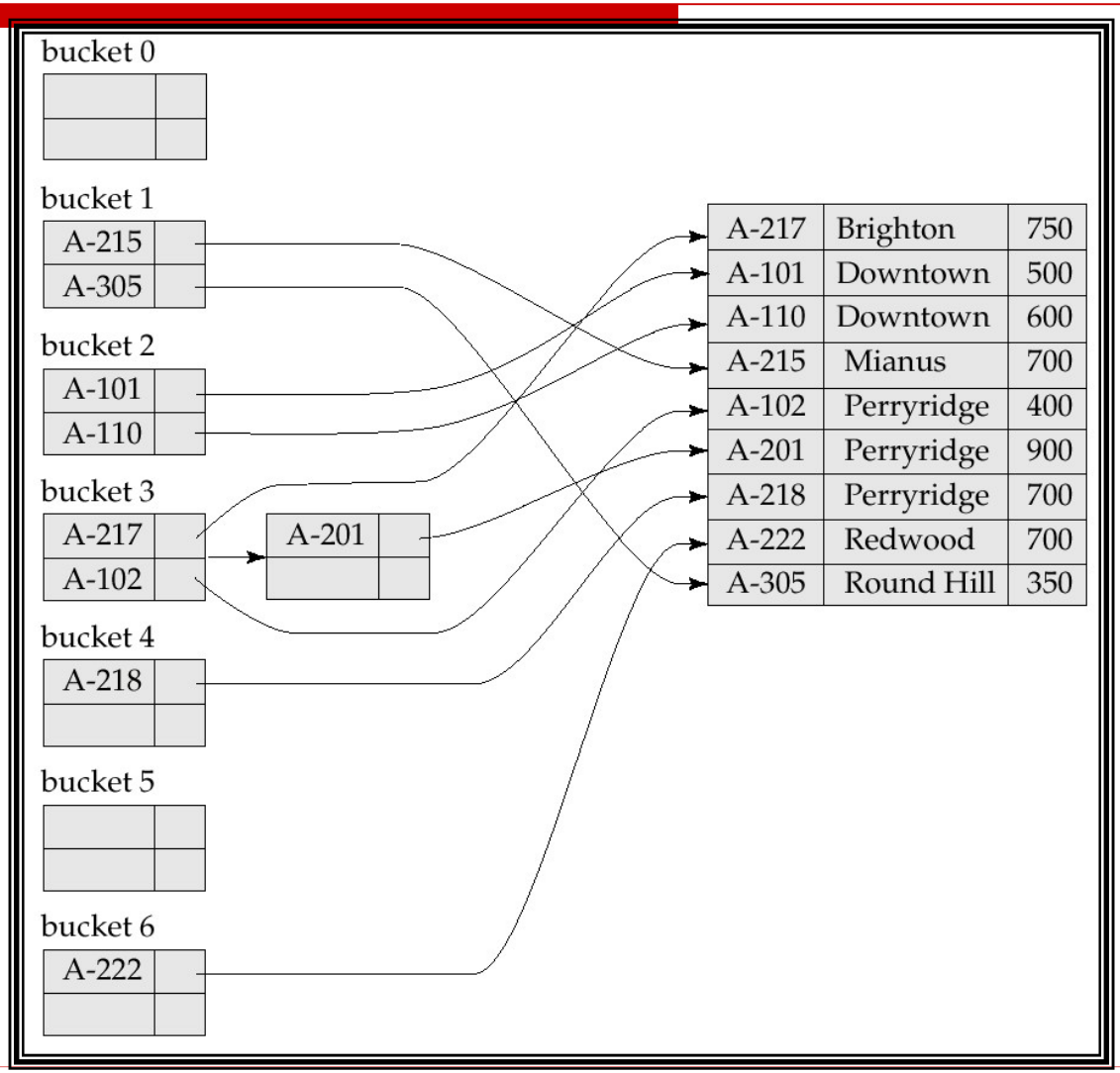
- 桶数目进行预留，即桶的数目选为 $(n_r/f_r)\times(1+d)$，（n_r：记录总数，f_r：一个桶能够容纳的记录数，d：避让因子）
- 为有桶溢出的桶分配溢出桶，如果溢出桶也满，则再分配溢出桶，形成溢出链。



散列索引

- 散列不仅适合用于文件组织，还适合用于索引结构的创建。
- 散列索引使将搜索码及其相应的指针组织成散列文件结构, 又称哈希索引。
- 具体方法：将散列函数作用于搜索码以确定对应的桶，然后将此搜索码及相应指针存入此桶(或溢出桶)中。
  - （1） **b＝Hash (k)**　　// **b:** 桶地址; **k:** 搜索码
  - （2） **b←point$_k$**　　// **point$_k$: k**的指针

account文件中在搜索码account-number上的散列索引



上述探讨散列的前提是数据库不变化，即静态散列，因此桶的数目也是固定。

但实际情况是数据库中记录是变化的，这一问题该如何解决？

目录

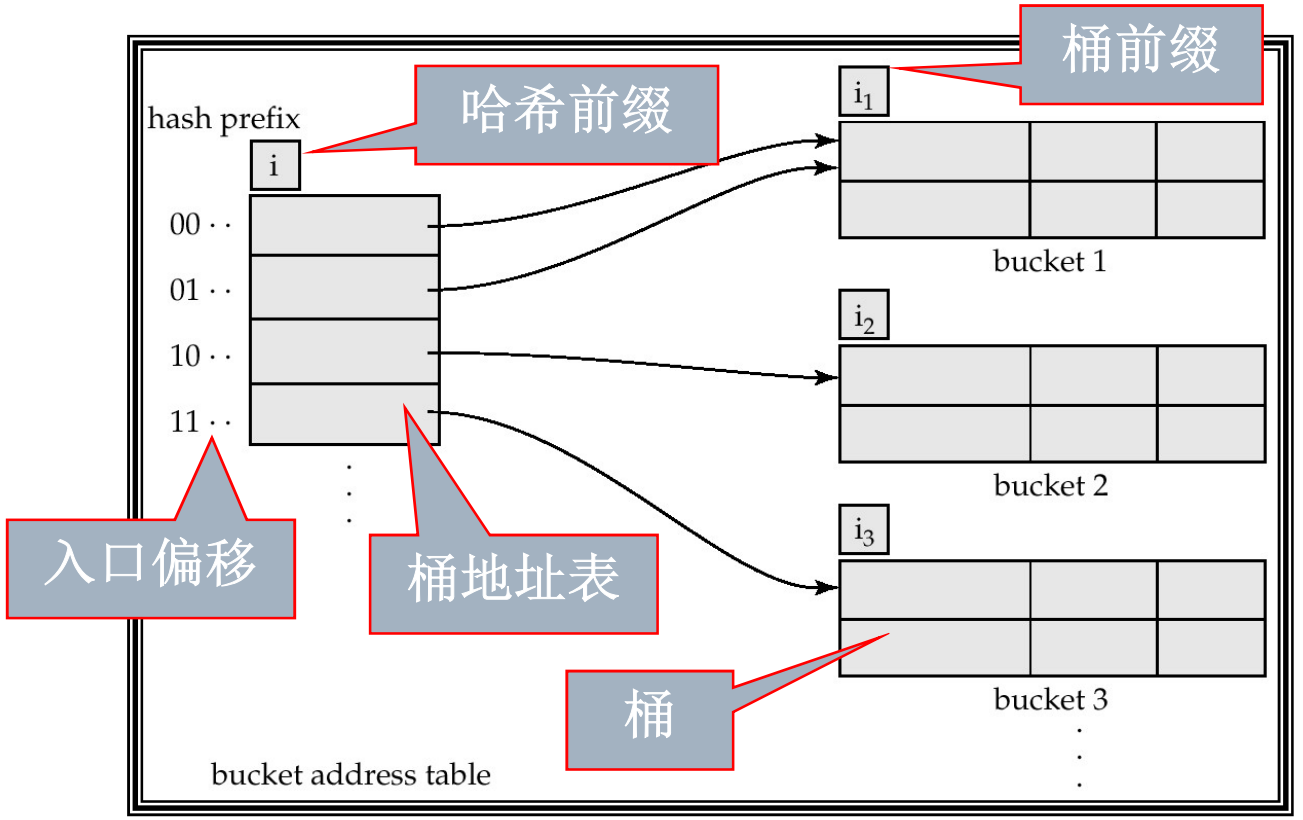
- 问题的提出
- 数据存储
- 索引基本概念
- 有序索引
  - 主索引与辅助索引
  - 稠密索引与稀疏索引
  - 多级索引
  - 索引更新
  - B^+树索引
- 散列
  - 静态散列
  - 动态散列
  - 顺序索引与散列比较
  - 位图索引（自学）

可扩充散列

- 支持动态变化的散列技术有：动态散列，可扩充散列，线性散列。这里介绍可扩充散列。
- 基本思想：桶的个数不固定，可以根据需要进行增加或减少。
  - 开始的时候，只有一个桶，一旦这个桶满，增加一个记录时，桶溢出并分割成两个桶。
  - 记录则基于其**Hash**值的最高位分布在两个桶中，最高位为**0**的记录在一个桶中，而最高位为**1**的记录分布在另一个桶中。
  - 即，通过对桶的分裂和合并来适应数据库大小的变化，保证空间的使用效率。

可扩充散列结构

- 可扩充散列的一般结构



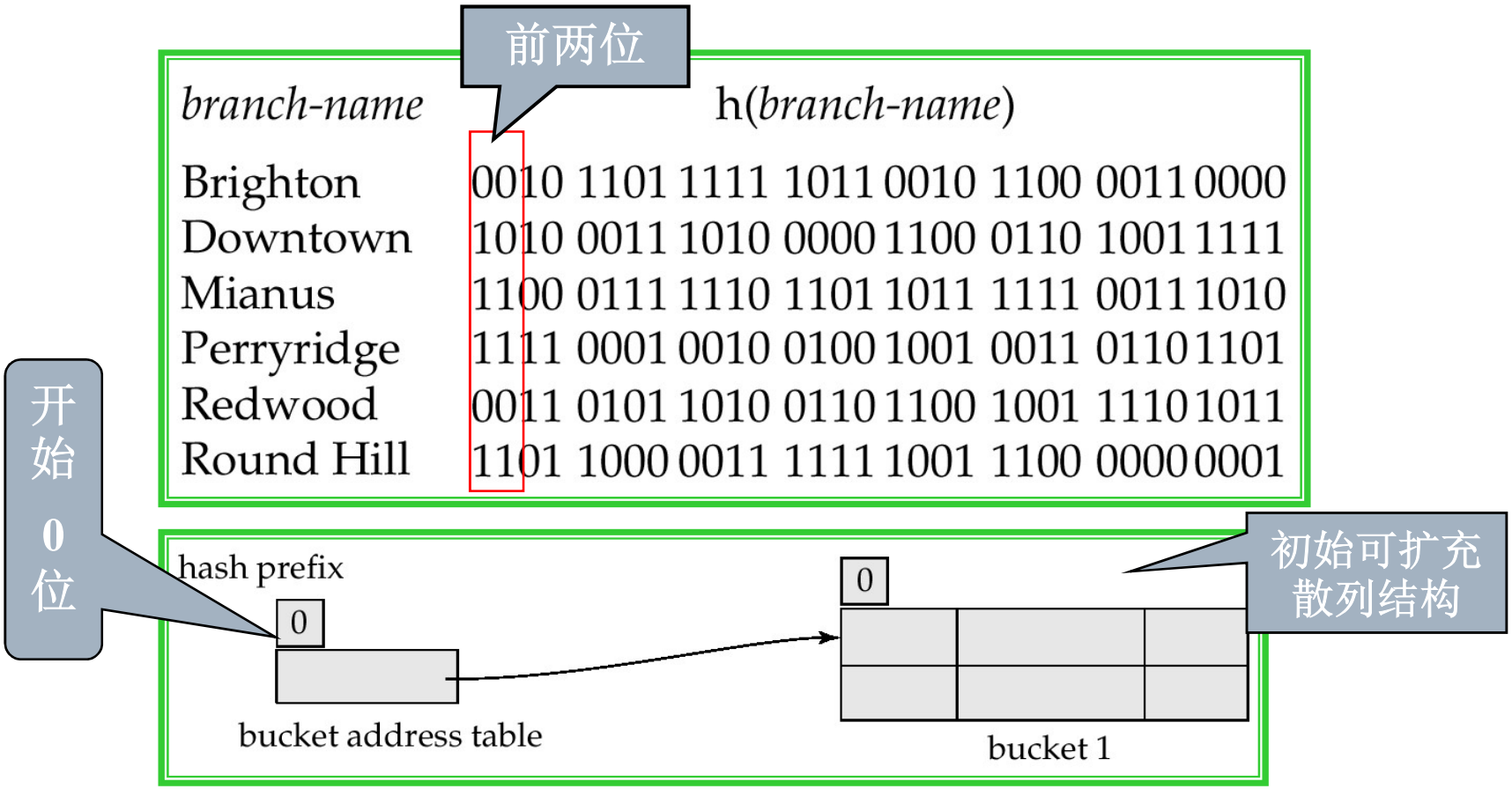
桶指针数组

- 通过散列函数生成b位的整数做桶地址，典型的b=32，那么可以建2^{32}超过了40亿个桶，除非特大型数据库，否则没有必要建这么多桶。
- 可以仅使用其中b位的一部分来构建桶指针数组。用一个数组作为目录的结构，数组中的元素存储桶地址。
- Hash(x)=101011….011，前i位定桶，初始i=0，桶多，i增加，2^i表示桶数量
- 例如：i=0，前0位定桶，2^0=1只有一个桶；i=1，前1位定桶，2^1=2有两个桶
- i的值随着数据的变化而变化，这样就实现了桶的动态扩充。
- 注意：桶地址表的入口偏移

Updates in Extendable Hash Structure

- 扩展散列的插入：
  - 自适应按需建桶，桶数够用，且尽量小。
  - 已有桶，有缝插针，无缝溢链，（不增加桶数）
  - 桶满，出现新Hash值，分裂-加桶，哪个满分哪个，不满的不分
  - 需量增至两桶 ，编为0 ，1号，用散列值的第1位
  - 增至 3 - 4桶 ，编为 00, 01, 10, 11号，用散列值的前2位定桶号， Hash值的前两位决定桶，重新分桶
  - 增至5 - 8 桶时，用3 位定桶号，….
  - 以此类推

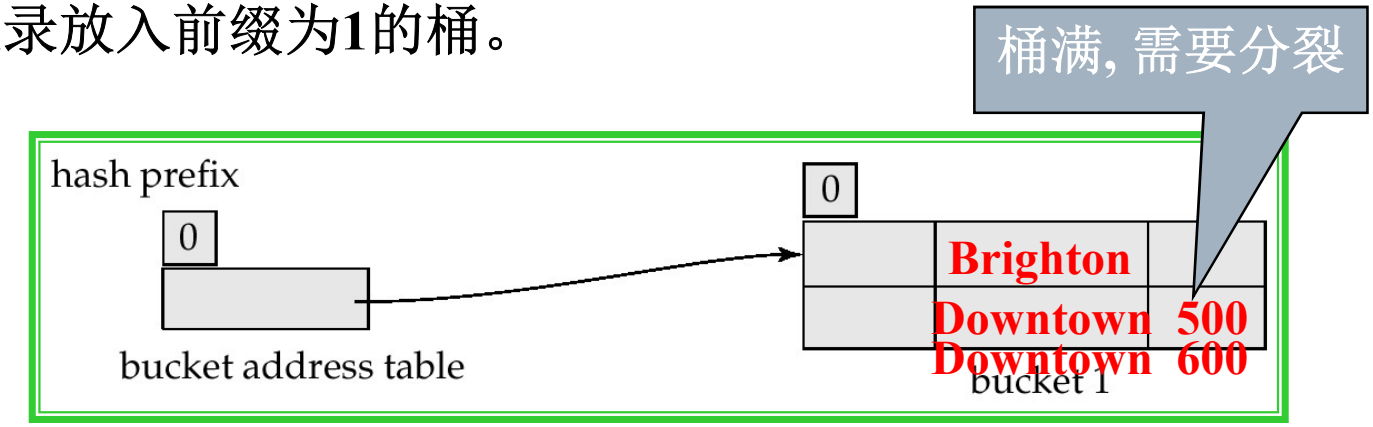
Example-插入更新



Initial Hash structure, bucket size = 2（假设）

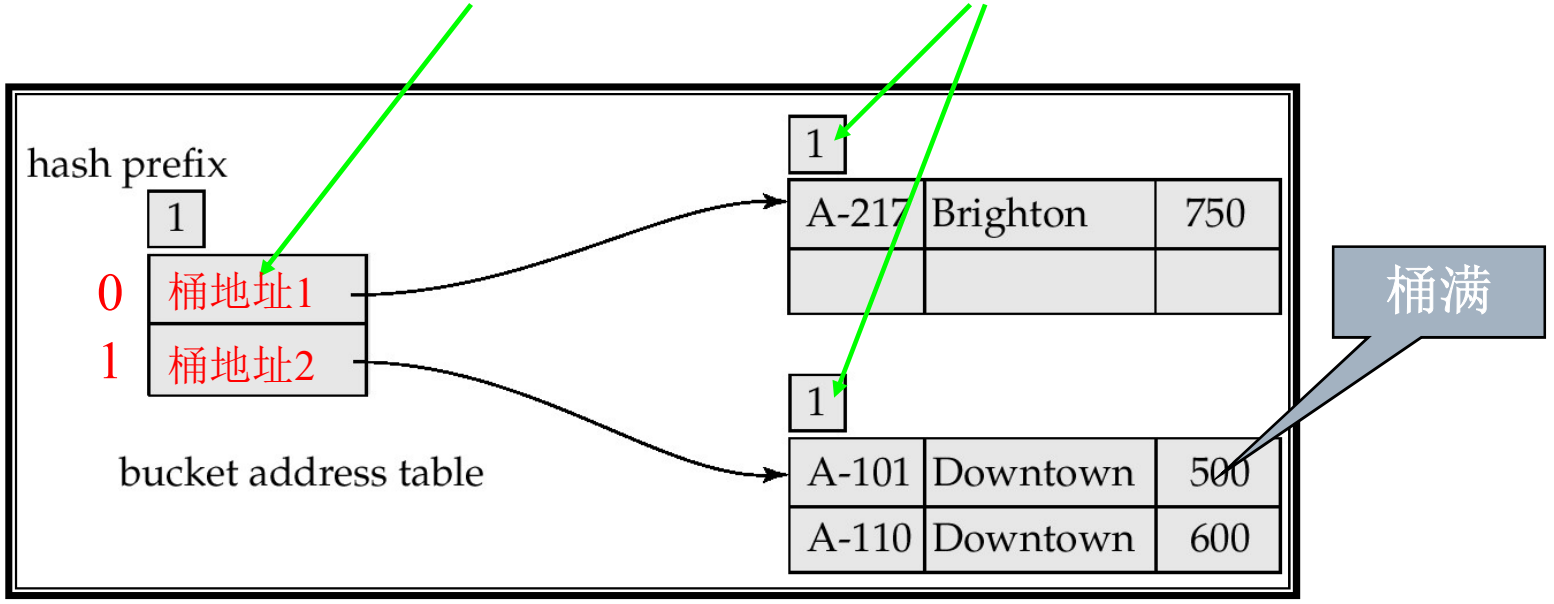
Example

- 初始桶可以放两个记录，当插入(A-217, Brighton, 750)，0位定桶，0位一定都相同！任意两条记录都将插入0号桶，系统插入该记录入初始桶。
- 接着，插入记录(A-101, Downtown, 500)，系统插入该记录入初始桶。
- 再插入(A-101, Downtown, 600)，桶满，h (Brighton)=0…，该记录在前缀（前1位）为0的桶不动， h (Downtown)=1…，该记录放入前缀为1的桶。



Example

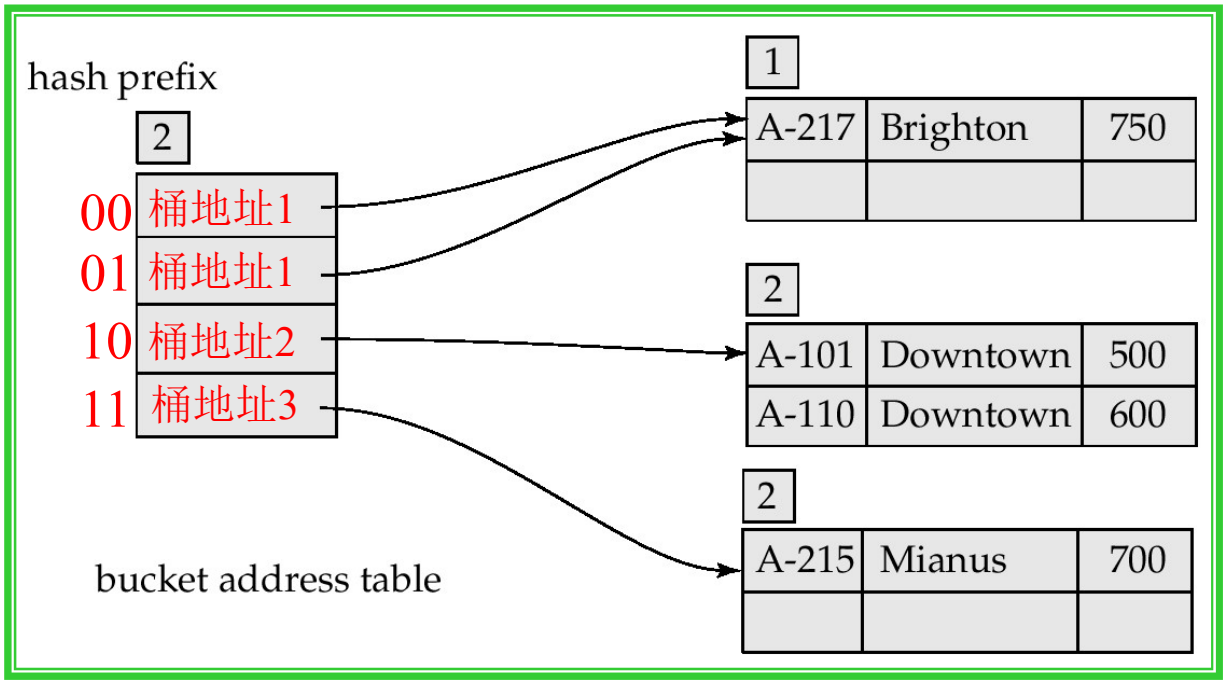
- Hash structure after insertion of one Brighton and two Downtown records 插入两个不同的散列值后，桶满
- 分裂为两桶，用1位定桶号，能区分两桶，1位定桶



- 前1位定桶，有0、1两个取值，即2^1=2个取值，有两个桶

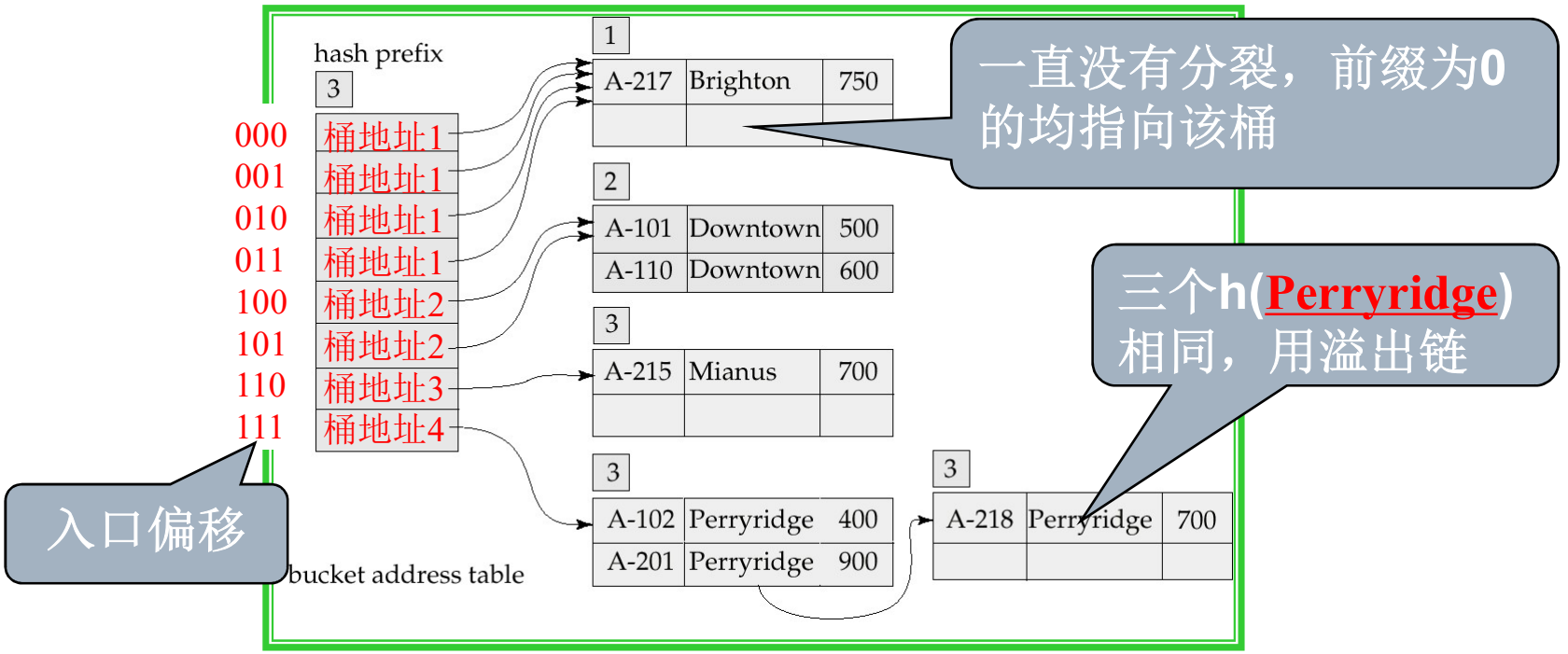
Example

- insertion of one Mianus record，h(Mianus)=1， 桶满（前页），$i=i_j$，i +1，桶分裂，i_j+1，有了3个不同Hash值，前2位定桶，可区分4 桶，够用条件下尽量节省空间，前缀为0的桶没有分裂，00、01都指向该桶



Example

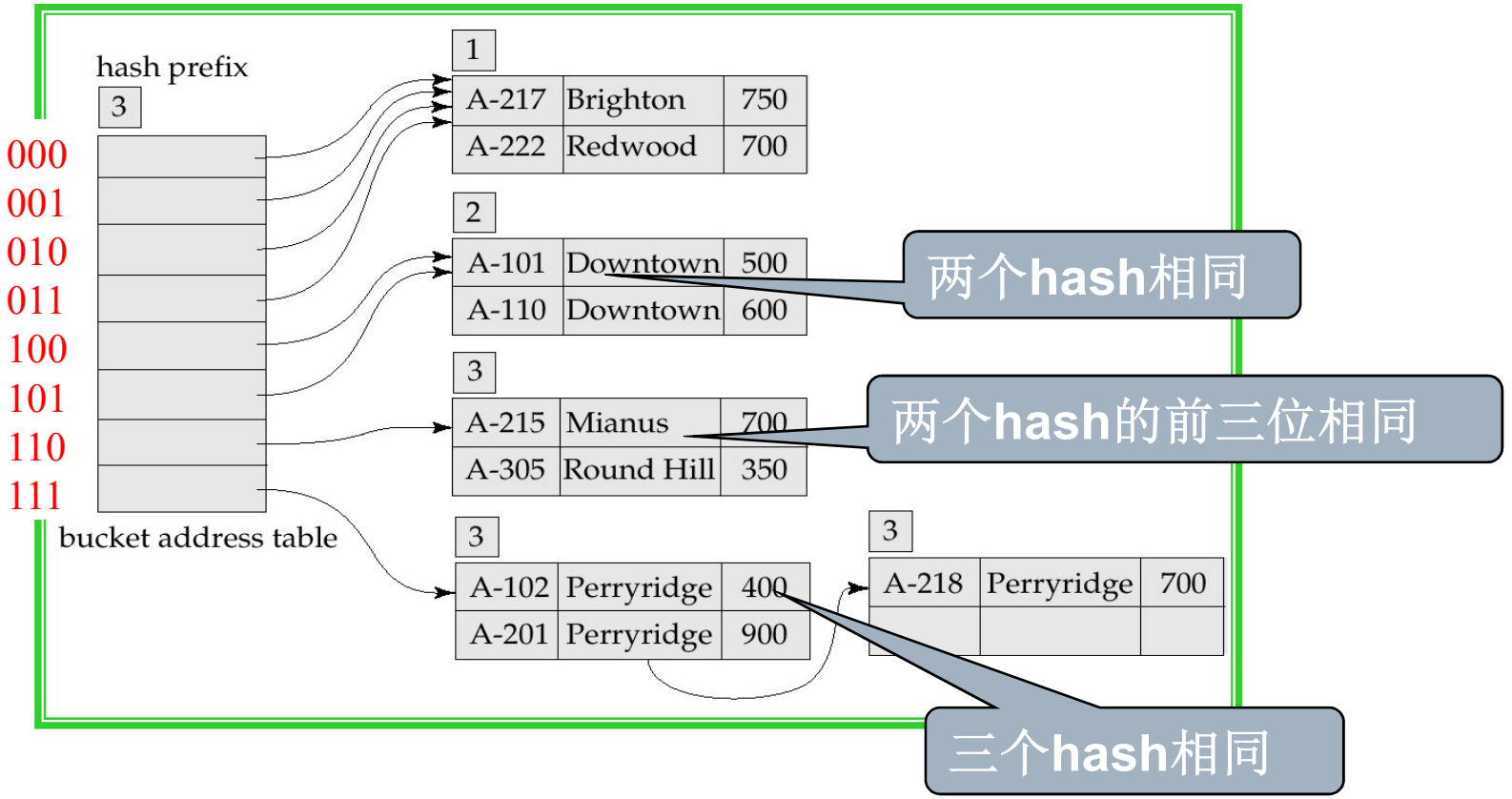
Hash structure after insertion of three Perryridge records



3位，可区分8 桶，现只4个不同Hash值，尚有余量

Example

- Hash structure after insertion of Redwood and Round Hill records



Query

- 按上述，i=3，3位定桶，桶地址依次为：000、001、010、011、100、101、110、111。
- 现有一个搜索码Redwood，计算h (Redwood)值为0011…11，由于i=3，取入口偏移001， 找桶地址表中的001号元素，存储的桶地址指向1号桶，在1号桶查“Redwood”，找到则有，否则无。
- 散列前缀、桶前缀均存储，元数据，随着桶的不断增加，前缀不断变化。

可扩展散列的特点

- 优点
  - 性能不退化，空间花销小
- 缺点
  - 查找多一个间接操作（散列前缀一层、桶一层，前缀计算）
  - 桶数可能最后很大，需用树结构记录
  - 加桶时花销大
- 可利用线性散列避免上述缺点

目录

- 问题的提出
- 数据存储
- 索引基本概念
- 有序索引
  - 主索引与辅助索引
  - 稠密索引与稀疏索引
  - 多级索引
  - 索引更新
  - B^+树索引
- 散列
  - 静态散列
  - 动态散列
  - 顺序索引与散列比较
  - 位图索引（自学）

有序索引与散列方案分析

- 有序索引适合 范围查询，散列索引适合 等值查询（值查询、点查询）。
- 各有优缺点，因此，数据库设计者需要考虑以下问题：
  - 索引或散列文件的周期性重组代价是否可以接受？
  - 插入和删除的相对频率如何？
  - 是否愿意增加最坏情况下的访问时间为代价优化平均时间？
  - 用户可能提出哪些查询？

目录

- 问题的提出
- 数据存储
- 索引基本概念
- 有序索引
  - 主索引与辅助索引
  - 稠密索引与稀疏索引
  - 多级索引
  - 索引更新
  - B^+树索引
- 散列
  - 静态散列
  - 动态散列
  - 顺序索引与散列比较
  - 位图索引（自学）

位图索引

- 位图索引实际上是掩码，或位置的布尔映射。
- 用于多搜索码查询，记录号有序，“小有序管大有序”，搜索码中的属性定义域不太大（“收入”就不合适），如
  - **E.g. gender（性别）, country（国家）, state（州）, …**
  - **E.g. 收入档次 income-level (income broken up into a small number of levels such as 0-9999(L1), 10000-19999(L2), 20000-50000(L3), 50000- infinity(L4))**
- 位图是位(bit)的数组

例子：位图索引

- 收入水平为L1的只有0号和2号，用（1 0 1 0 0）表示。

0号 3号 → 1 0 0 1 0

0号 2号 → 1 0 1 0 0

- 男性只有0号和3号，用（1 0 0 1 0）表示。

record number	*name*	*gender*	*address*	*income-level*
0	John	m	Perryridge	L1
1	Diana	f	Brooklyn	L2
2	Mary	f	Jonestown	L1
3	Peter	m	Brooklyn	L4
4	Kathy	f	Perryridge	L3

Bitmaps for *gender*

m	1 0 0 1 0
f	0 1 1 0 1

Bitmaps for *income-level*

L1	1 0 1 0 0
L2	0 1 0 0 0
L3	0 0 0 0 1
L4	0 0 0 1 0
L5	0 0 0 0 0

例子：位图索引-查询

- 根据上述关系及索引，考虑查询：收入在10000～19999之间的女性。关系代数表达式为：$\sigma_{gender=f \wedge income\_level=L2}(r)$
- gender = f的位图为（01101），income_level=L2的位图为（01000），计算（01101）与（01000）的交（按位与(and)运算），结果为01000，定位记录号为2 。
- 位图索引支持位运算and ， or， not

位图索引应用

- **Bitmap indices**在多属性检索时有用，对单属性检索意义不大，且适用于数据分析应用中的，经过泛化的数据，如前面例子。
- **Bitmap indices**占空间小，一个记录一位，如上面例子中，**income_level**索引共**25**位，约**3**个字节，如果记录文件一个元组为**50**个字节，整个关系为**250**个字节，那么，索引占整个关系的**1.2%**
- 使用位图索引有时甚至不需要访问数据文件，如查询性别为女性的记录，直接计算索引即可。
- 当删除记录时，位图索引更新要注意，通过一个存在位图（和索引配合使用进行跟新）实现（自学）。

SQL中的索引定义

- 注意：SQL标准并未为数据库用户或管理员提供任何在数据库系统中控制创建和维护何种索引的方法，但多数DBMS允许用户通过DML的命令对索引创建和删除进行控制。
- 创建索引的命令：
  - **create index <index-name> on <relation-name> (<attribute-list>)**
  - **e.g. create index *b-index* on *borrower (customer_name, loan_number)*** 组合索引
  - 很多数据库系统还提供一些方法来详细说明使用的索引类型(如散列或B^+树)。

产品支持

- 主流的数据库产品支持的主要索引类型情况：
  - **SQL Server 2005：B树索引**
  - **Oracle 10g： B树索引，位图索引，哈希索引（划分）**
  - **PostgreSQL：B树索引，哈希索引，多码索引， R树索引**
  - **DB2： B^+树索引**
- 以上仅列出主要的索引类型支持，注意有些索引实质上是上述基本索引派生出来的，另外注意，有些产品说明中所说的支持**B树索引**，实质上是**B^+树索引**。

小结

- 散列适用于大型数据库的文件组织
- 利用散列函数实现了记录定位效率与记录数量的无关性，是一个很大进步
- 静态散列适合那种查询频繁而无数据更新的情况
- 动态散列适用于数据经常更新的数据库
- 散列索引适合等值查询，不适合比较查询

课堂随测

- 1、既然索引可以加快查询处理速度，是否应该对所有搜索码建立索引？
- 2、什么情况下使用稠密索引，什么情况下使用稀疏索引？

- 答案写在一张纸上，并写上姓名、学号，下课前交。