第14讲索引和哈希
(14章)

关继宏教授
电子邮件:jhguan@tongji.edu.cn
计算机科学与技术系
同济大学

课程大纲

- **第0部分:概述**
  - Ch1:介绍
- **Part 1关系数据库**
  - Ch2:关系模型(数据模型，关系代数)
  - Ch3&4: SQL(结构化查询语言)
  - Ch5:高级SQL
- **第二部分数据库设计**
  - Ch6:基于E-R模型的数据库设计
  - Ch7:关系型数据库设计
- **第三部分:应用程序设计与开发**
  - Ch8:复杂数据类型
  - Ch9:应用开发
- **Part 4大数据分析**
  - Ch10:大数据
  - Ch11:数据分析

第5部分数据存储和索引

- Ch12:物理存储系统
- Ch13:数据存储结构
- **Ch14:索引**

- **第6部分:查询处理与优化**
  - Ch15:查询处理
  - Ch16:查询优化
- **第7部分事务管理**
  - Ch17:交易
  - Ch18:并发控制
  - Ch19:恢复系统
- **第8部分:并行和分布式数据库**
  - Ch20:数据库系统架构
  - Ch21-23:并行和分布式存储，查询处理和事务处理
- **第9部分**
  - **DB平台:OceanBase、MongoDB、Neo4J**

大纲

<s:1>基本概念

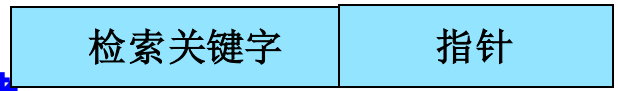
- 有序索引
- B+ -tree & B-tree索引
- 静态和动态哈希
- 有序索引与哈希
- SQL中的索引定义
- 多个键访问

索引与数据()

- 索引可提高检索效率,其结构(二叉树,B +树等)占用空间小,所以访问速度快
  - 如果表中的一条记录在磁盘上占用 1000 字节， 对其中 10 字节 的一个字段建立 索引， 那么该记录对应的索引项的大小只有 10 字节 。如SQL Server的最小空间分配单元是“页页面” ,一个页在磁盘上占用8 k空间,可以存储上述记录8条,但可以存储索引800条
  - 从一个有8000条记录表中检索符合某个条件的记录,如没有索引,可能需要遍历8000条×1000字节/ 8 k字节= 1000个页面才能找到结果。如果在检索字段上有上述索引,则可以在8000条×10字节/ 8 k字节= 10个页面中检索到满足条件的索引块,然后根据索引块上的指针逐一找到结果数据块,这样的I / O访问量要少很多

基本概念

- **索引机制**
  - 加快对所需数据的访问
  - 索引文件通常比原始文件小得多
- **搜索键(/)**
  - 用于查找文件/表中的记录的一组属性
  - 索引文件由以下形式的记录(称为索引项)组成(search-key, pointer)
- **有两种索引**
  - **有序索引():搜索键按排序顺序存储**
  - **哈希索引():使用"哈希函数"将搜索键均匀分布在"桶"上**



指标评价指标

- 高效支持的访问类型
  - Equal-query()、Range-query()
  - 能有效支持的访问类型,如找到具有特定属性值的所有记录,找到其属性值在某个特定范围所有记录(等于查询,查询范围)
- 访问时间:
- 插入时间:插入一个新数据项时间,包括:找到插入位置时间+更新索引结构时间
- 删除时间:删除一个数据项时间,包括:找到待删除项时间+更新索引结构时间
- 空间开销:

大纲

- 基本概念

有序索引

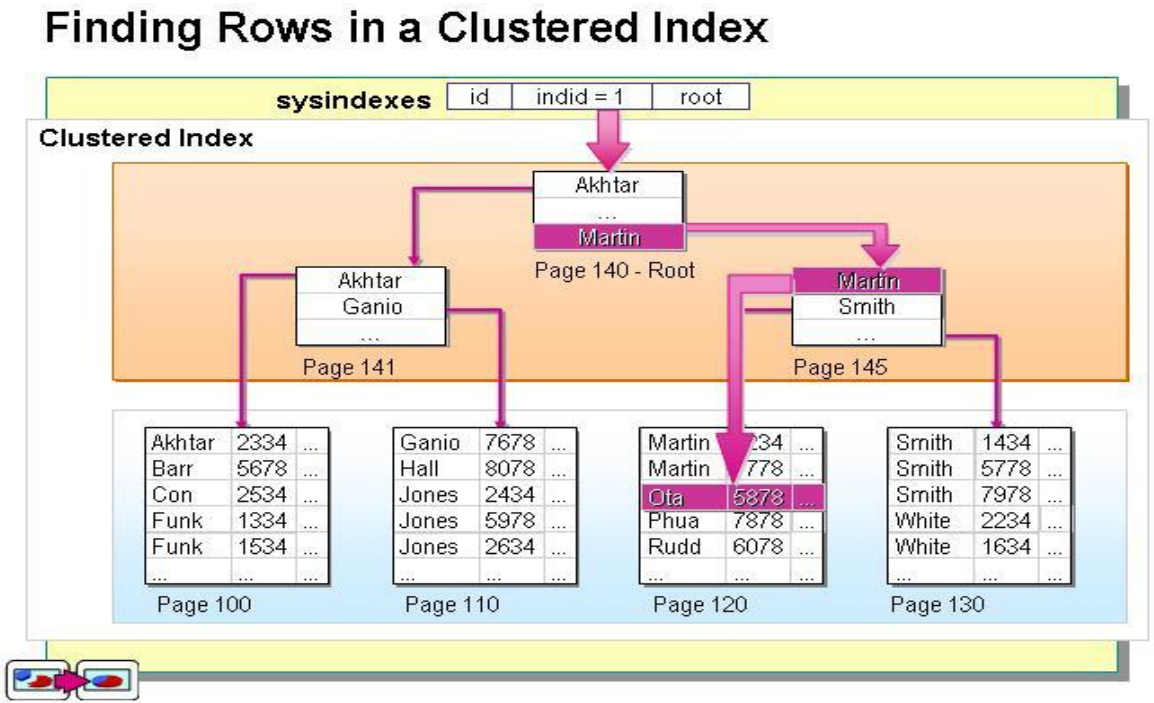
- B+ -tree & B-tree索引
- 静态和动态哈希
- 有序索引与哈希
- SQL中的索引定义
- 多个键访问

有序索引

- **命令指数**
  - 索引项按搜索键值排序
  - 主索引和次索引
  - **一级索引()，聚类索引**
    - 包含记录的文件按某个搜索码指定的顺序排序,该搜索码对应的索引称为群集索引
  - **二级索引()，无聚类索引()**
    - 一种索引，其搜索键指定的顺序与文件的顺序不同
- **索引顺序文件()**
  - 具有主索引的有序顺序文件
  - 索引顺序文件 是 顺序文件 的扩展，其中各记录本身在介质上也是顺序排列的，包含了直接处理和修改记录的能力。索引顺序文件能像顺序文件一样进行快速顺序处理，既允许按物理存放次序（记录出现的次序），也允许按逻辑顺序（由记录主关键字决定的次序）进行处理。索引顺序文件通常用树结构来组织索引。静态索引结构ISAM和动态索引结构VSAM

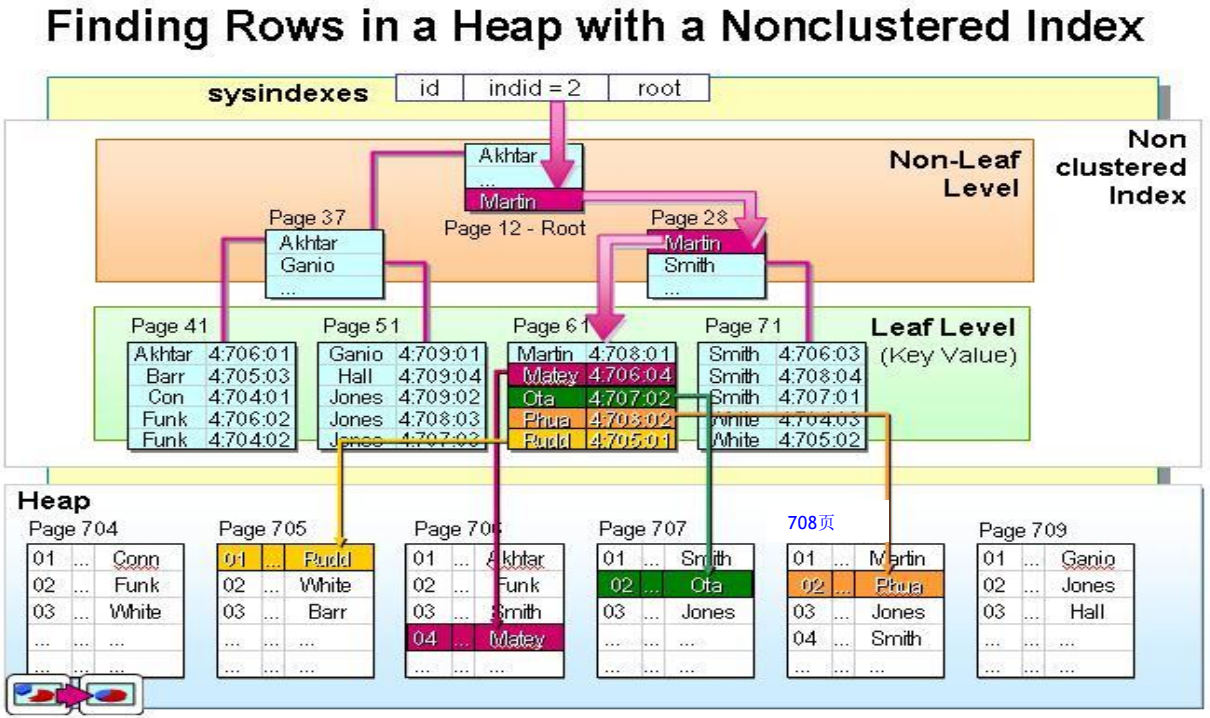
Primary Index:聚类索引(Clustering Index)

- 聚集索引的叶节点就是数据节点 ， 索引顺序就是数据物理存储顺序 。 一个表最多只能有一个 聚集索引



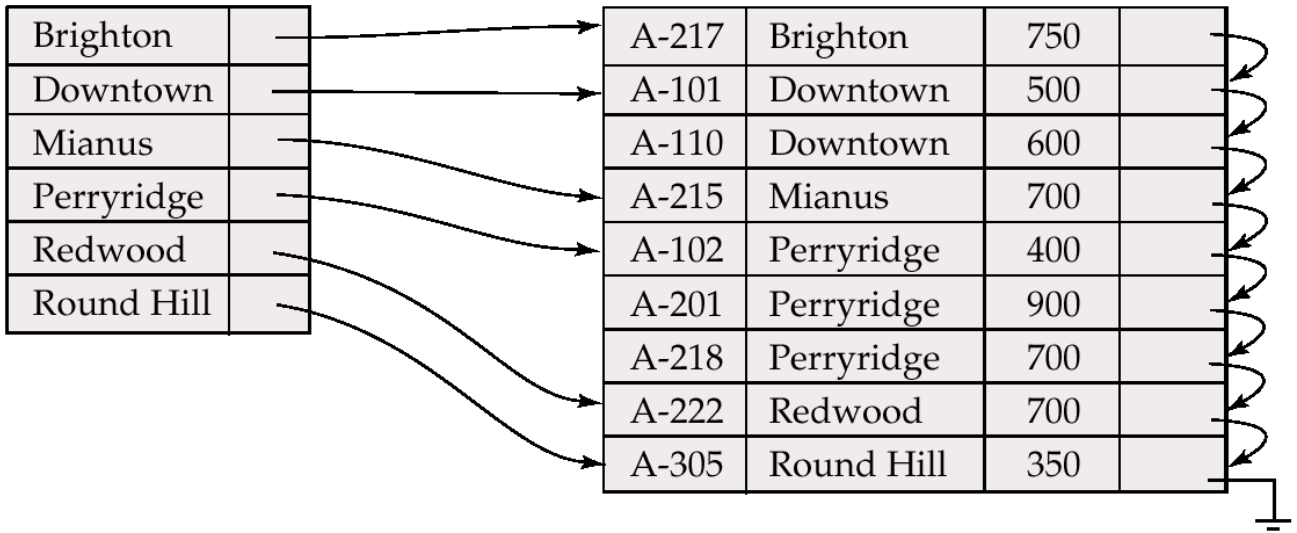
二级指数:非聚类指数

- 非聚集索引的叶节点仍然是索引节点， 有一个指针指向对应的数据块 。 非聚集索引顺序与数据物理排列顺序无关



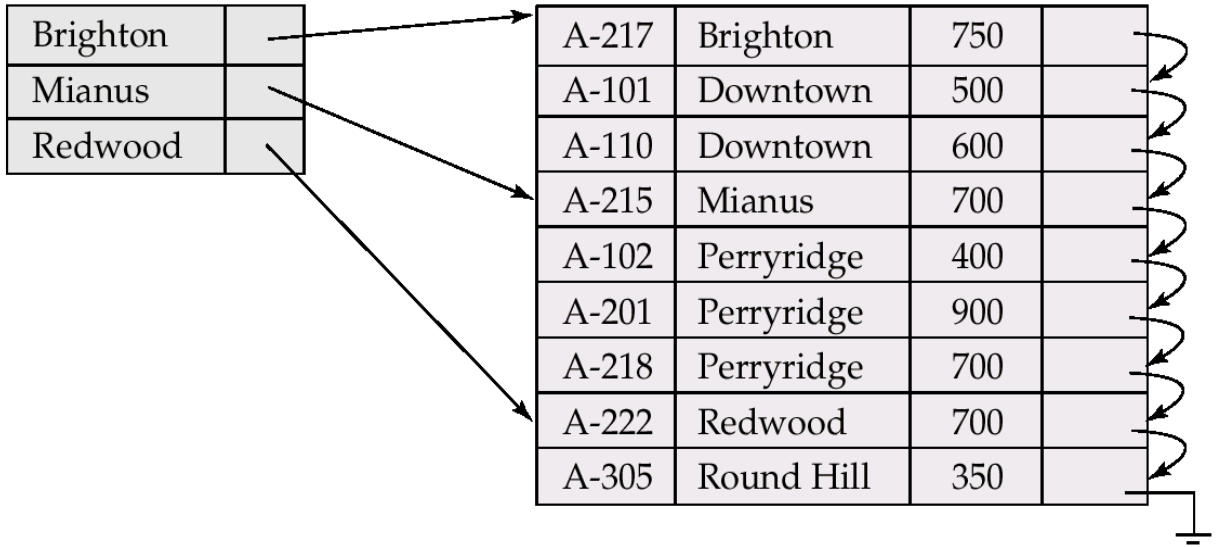
稠密索引

- **密集指数(稠)**
  - 为文件中的每个搜索键值显示索引记录



稀疏索引

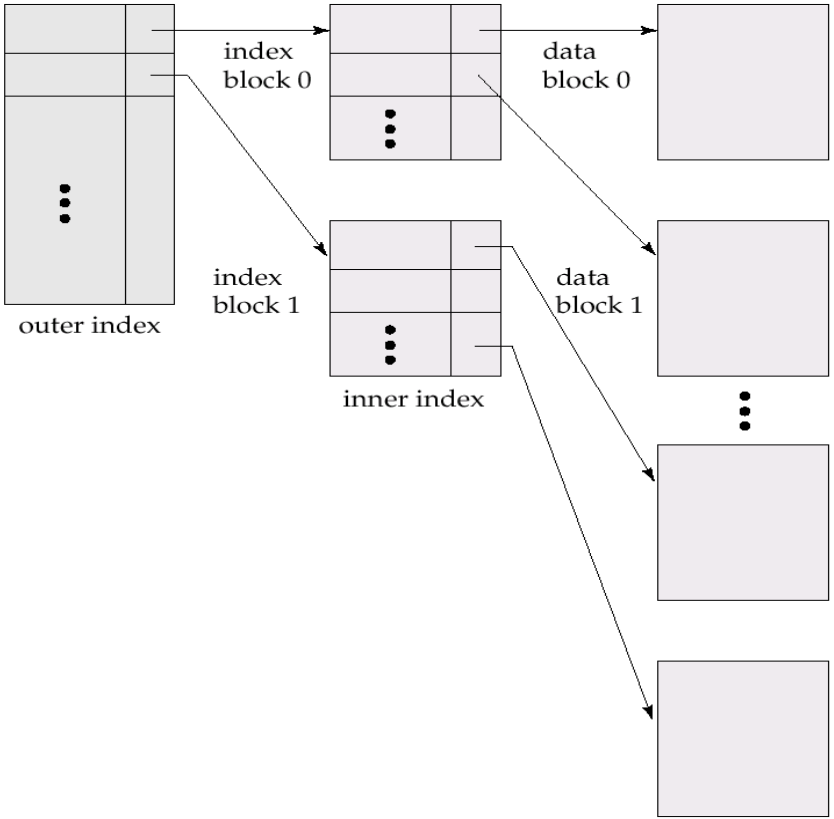
- **稀疏索引()**
  - 当记录按搜索键顺序排序时，只包含一些搜索键值的索引记录(为什么?)



多级索引()

- 如果主索引不适合内存，则数据访问变得昂贵
- 为了减少对索引记录的磁盘访问次数，可以将主索引视为一个顺序文件，并在其上构造一个稀疏索引
  - **外索引——主索引的一个稀疏索引**
  - **内索引——主索引文件**
- 如果外部索引太大，无法在主存中容纳，还可以创建另一个级别的索引，以此类推

多级索引(续)

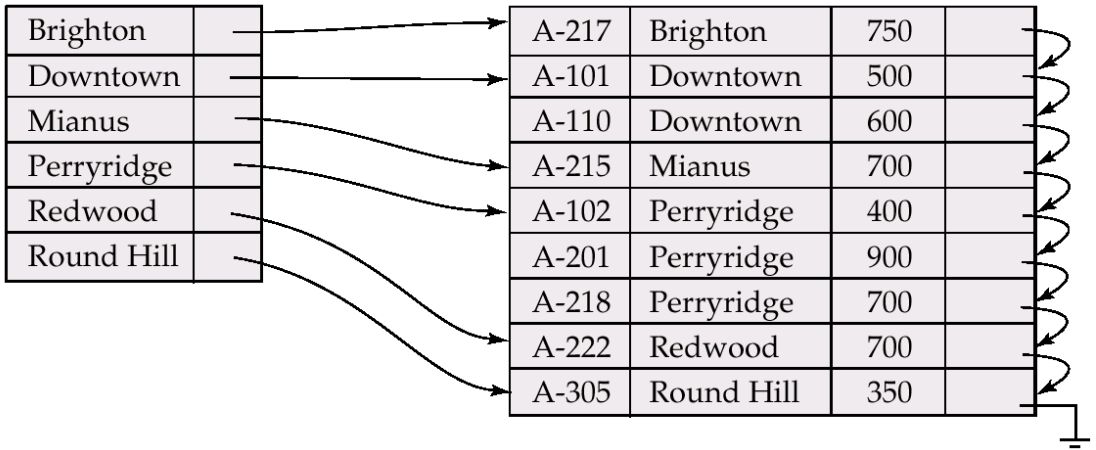

index
block 0
data
block 0
index
block 1
data
block 1
outer index
inner index

密集索引与稀疏索引

- To locate a record with search-key value K:
  - **Dense index**
    - Find index record with **search-key value = K**
  - **Sparse index**
    - Find index record with **largest search-key value <= K**
    - Search file sequentially starting at the record to which the index record points
  - **Sparse index** is generally slower than **dense index** for locating records but saves more storage space
  - **Space and maintenance for insertions and deletions**

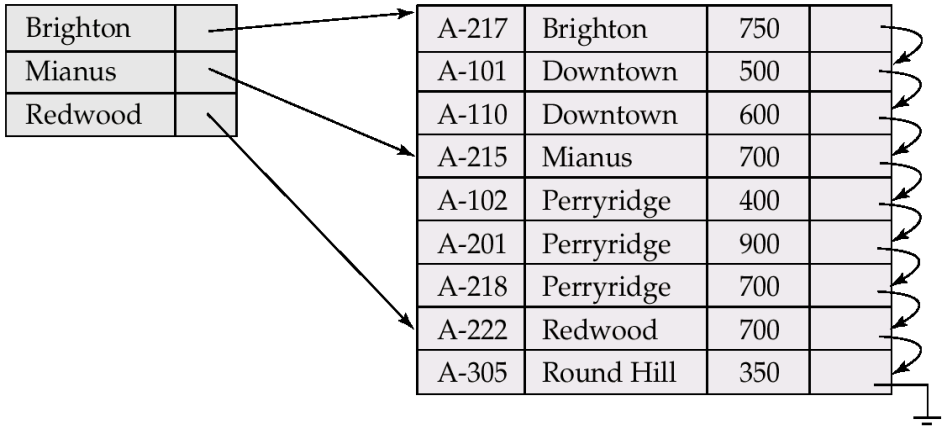
索引更新:删除

- **单级索引删除**
  - **密集索引——索引中搜索键的删除类似于文件记录的删除**



索引更新:删除

- **单级索引删除**
  - **稀疏索引**
    - 如果索引中存在搜索键的条目，则通过将索引中的条目替换为文件中的下一个搜索键值来删除该条目
    - 如果下一个搜索键值已经有索引项，则删除该项而不是替换该项



索引更新:插入

- **单级索引插入**
  - 使用搜索键值执行查找
  - **密集索引-如果搜索键值没有出现在索引中，则插入它**
  - **稀疏索引——如果索引为文件的每个块存储一个条目，除非创建一个新的块，否则不需要对索引进行更改。在这种情况下，出现在新块中的第一个搜索键值被插入到索引中**
- **多级插入/删除**
  - 单级算法的扩展

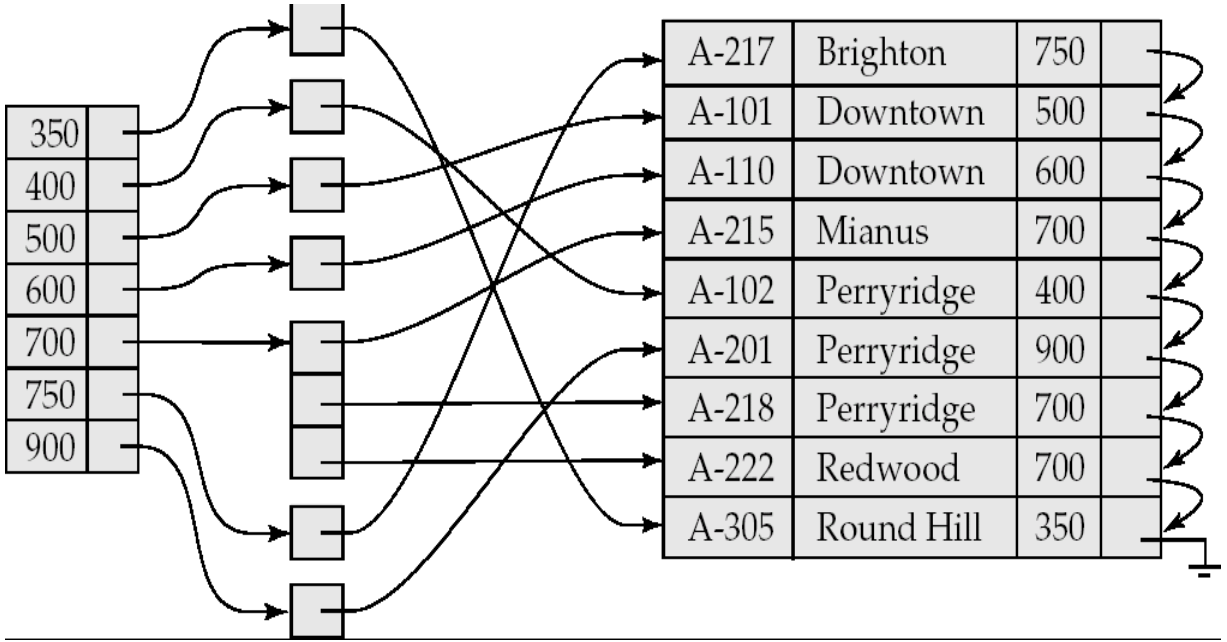
Dense vs. Sparse Index

- 插入和删除的空间和维护
  - 稀疏索引对于插入和删除需要更少的空间和更少的维护开销
  - 良好的权衡:稀疏索引对文件中的每个块都有一个索引条目，对应于块中最小的搜索键值

二级指标

- **查找值在一定范围内的所有记录**
  - **例1:在按账号顺序存储的账户关系中，我们可能想要找到某一特定支行的所有账户**
  - **例2:查找具有指定余额或余额范围的所有账户**
- **二级索引**
  - 为每个搜索键值建立一个索引记录的二级索引
  - 索引记录指向一个存储桶，该存储桶包含指向具有该特定搜索键值的所有实际记录的指针

账户余额字段的二级索引



桶

一级和二级索引

- **二级索引必须是密集的(为什么?)**
- 当一个文件被修改时，文件上的每个索引都必须更新。更新索引会增加数据库修改的开销
- **使用主索引的顺序扫描是高效的，但是使用二级索引的顺序扫描是昂贵的**
  - 每次记录访问都可能从磁盘获取一个新的块

大纲

- 基本概念
- 有序索引

→B + -tree & B-tree索引

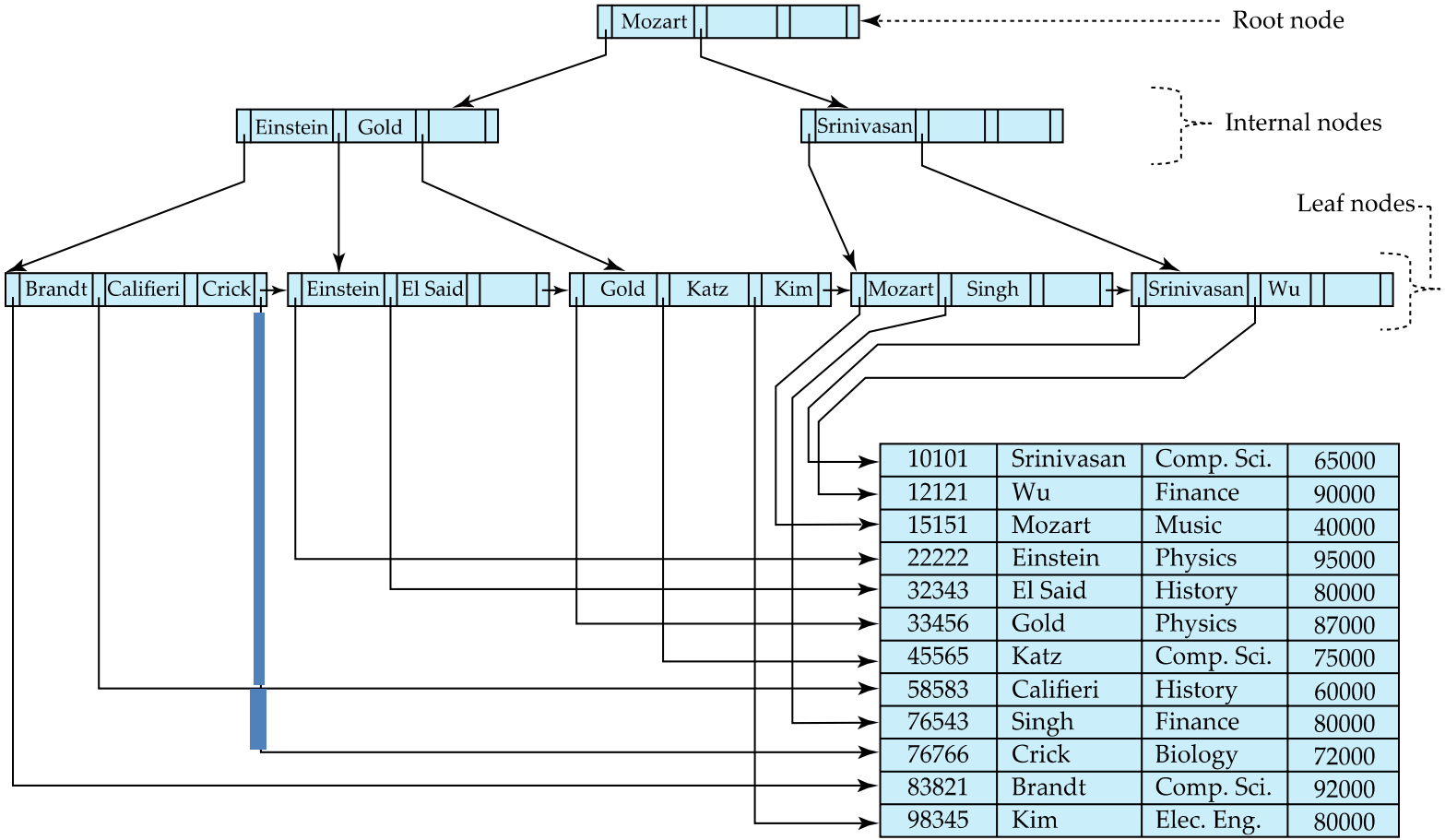
- 静态和动态哈希
- 有序索引与哈希
- SQL中的索引定义
- 多个键访问

B+ -树索引文件

B+ -tree是索引顺序文件的替代方案

- **索引顺序文件的缺点**
  - 性能随着文件的增长而下降，因为创建了许多溢出块()。需要对整个文件进行周期性的重组
- **B+ -树索引文件**
  - **优点:在面对插入和删除时，通过小的和局部的改变自动重组自身。不需要对整个文件进行重组来保持性能**
  - **缺点:额外的插入和删除开销，空间开销**
  - B+树被广泛使用，因为它的优点大于缺点

B +树的例子



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

B+ -Tree索引文件(续)

- Typical B+-tree node

  - K_i are the search-key values. The search-keys in a node are ordered, i.e.,

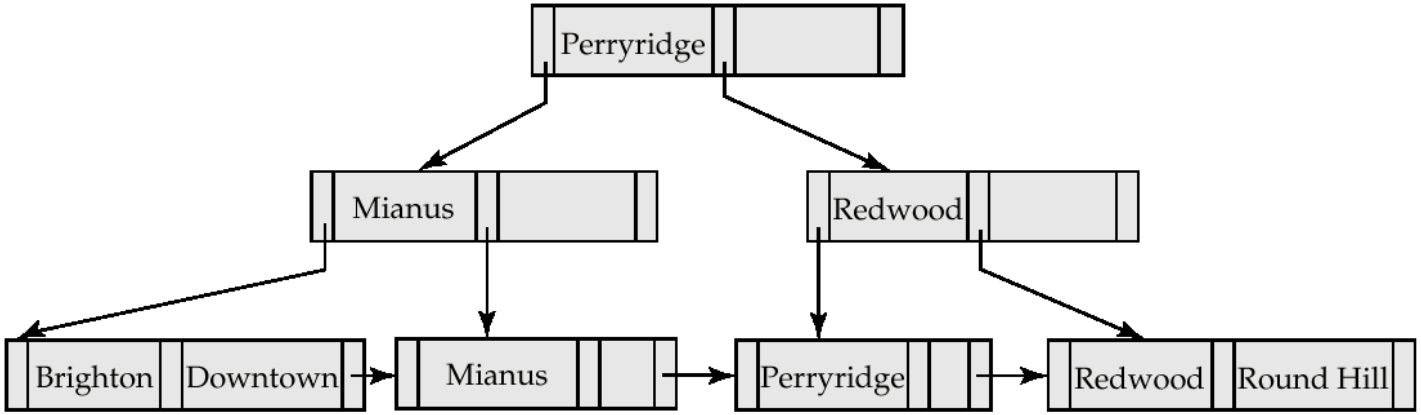
$$K_1 < K_2 < K_3 < \cdots < K_{n-1}$$

  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes)

B+ -Tree Index Files (Cont.)

- A B+-tree is a rooted tree (有根树) satisfying the following properties:
  - B+-tree is a **balanced tree** and all the paths from root to leaf nodes are of the same length
  - **Internal node**
    - Each node has between $\lceil n/2 \rceil$ **and** n **children (pointers)**
  - **Leaf node**
    - Each node has between $\lceil (n-1)/2 \rceil$ **and** $n-1$ **values**
  - **Root node**
    - If the root is not a leaf, it has **at least 2 children**
    - If the root is a leaf (i.e., there are no other nodes in the tree), it can have between **0 and n–1 values**

B +树的例子



账户文件的B+ -树(n = 3)

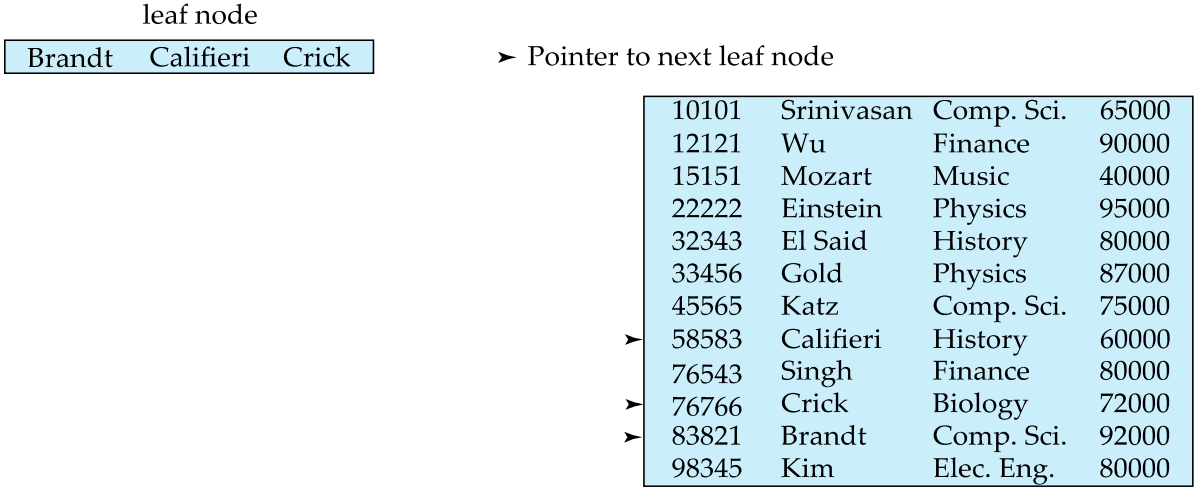
- Leaf nodes must have between 1 and 2 values ($\lceil (n-1)/2 \rceil$ and $n-1$)
- Non-leaf nodes other than root must have between 2 and 3 children ($\lceil n/2 \rceil$ and n)
- Root must have at least 2 children

B + -Tree中的叶节点

- **Properties of a leaf node**

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

  - 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 the search-key does not form a primary key (**why?**)
  - P_n **points to next leaf node in search-key order**



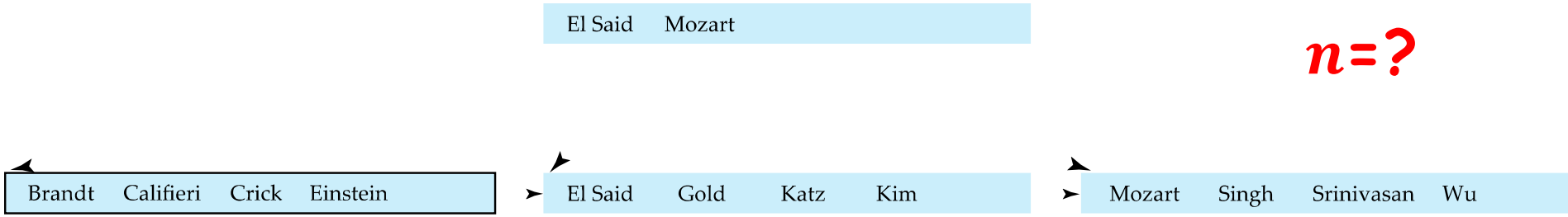
$n=4$

指导文件

B + -Tree中的非叶节点

- **Non leaf nodes** form a **multi-level sparse index** on the leaf nodes. For a non-leaf node with n pointers:
  - All the search-keys in the subtree to which P_1 **points are less than** K_1
  - For $2 \leq i \leq n-1$, 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_i
  - All the search-keys in the subtree to which P_n **points are greater than or equal to** K_{n-1}

B + -tree的例子



- B+-tree for ***instructor*** file ($n = 6$)
  - Leaf nodes must have between **3** and **5** values (**$\lceil (n-1)/2 \rceil$ and $n-1$**)
  - Non-leaf nodes other than root must have between **3** and **6** children (**$\lceil n/2 \rceil$ and n**)
  - Root must have at least **2** children

关于B + -tree的观察

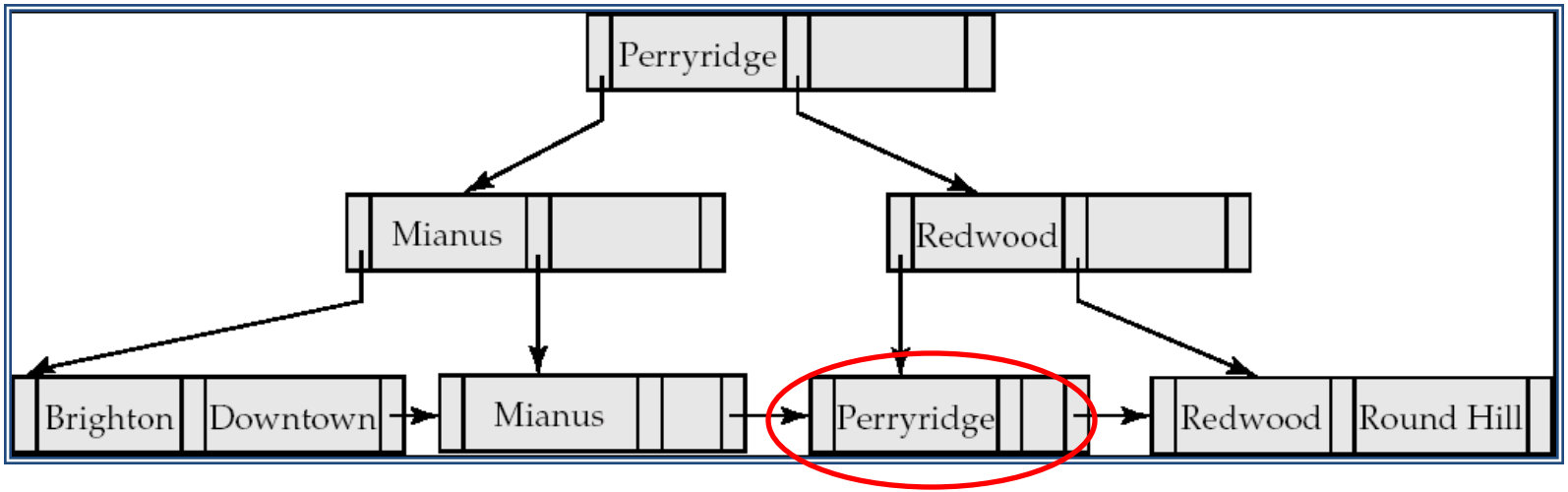
- Since the **inter-node** connections are achieved by **pointers**, "logically" close blocks need not be "physically" close
- The non-leaf levels of the B⁺-tree form a hierarchy of sparse indices
- The **B⁺-tree** contains a relatively small number of levels, and search can be conducted efficiently
  - **If there are K search-key values in the file, the tree height is no more than $\lceil log_{n/2}(K) \rceil$**
    - Level below root has at least $2 * \lceil n/2 \rceil$ **values**
    - Next level has at least $2 * \lceil n/2 \rceil * \lceil n/2 \rceil$**values**
    - …
- Insertions and deletions to the index file can be handled efficiently

关于B +树的查询

- **Find all records with a search-key value of k**
  - Start with the root node
    - Check the node for the **smallest search-key value > k**
    - If such a value exists, assume that it is K_i. Then follow P_i to the child node
    - Otherwise $k \geq K_{n-1}$, where there are n pointers in the node. Then follow P_n to the child node
  - If the node reached by following the pointer above is not a leaf node, repeat the above procedure on the node, and follow the corresponding pointer
  - Eventually reach a leaf node. If for some i, key $K_i = k$, follow pointer P_i to the desired record or bucket. Else no record with search-key value k exists

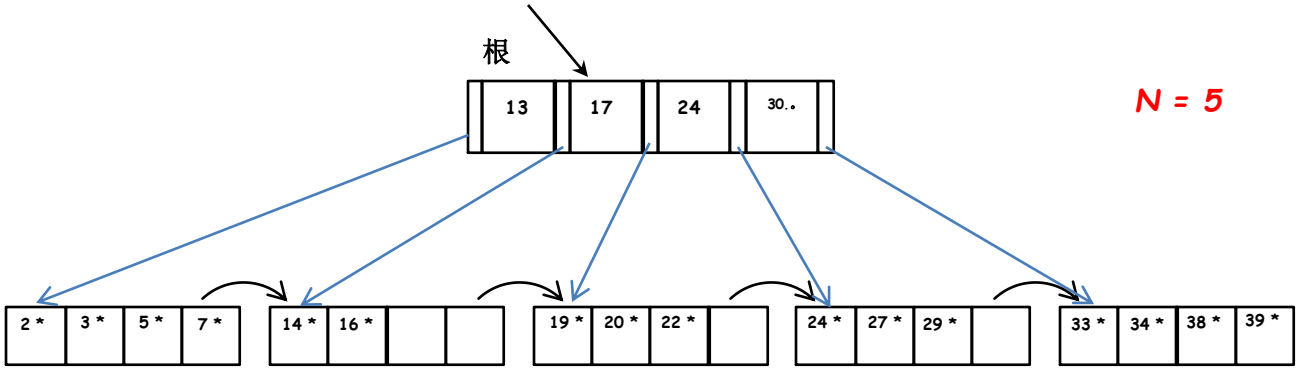
示例:查询B + -Tree

- 搜索从根开始，键比较指向叶节点
  - 搜索Perryridge



示例:在B + -Tree上查询

- 搜索从根开始，键比较指向叶节点
  - 搜索5*，15*，所有数据项>= 24*



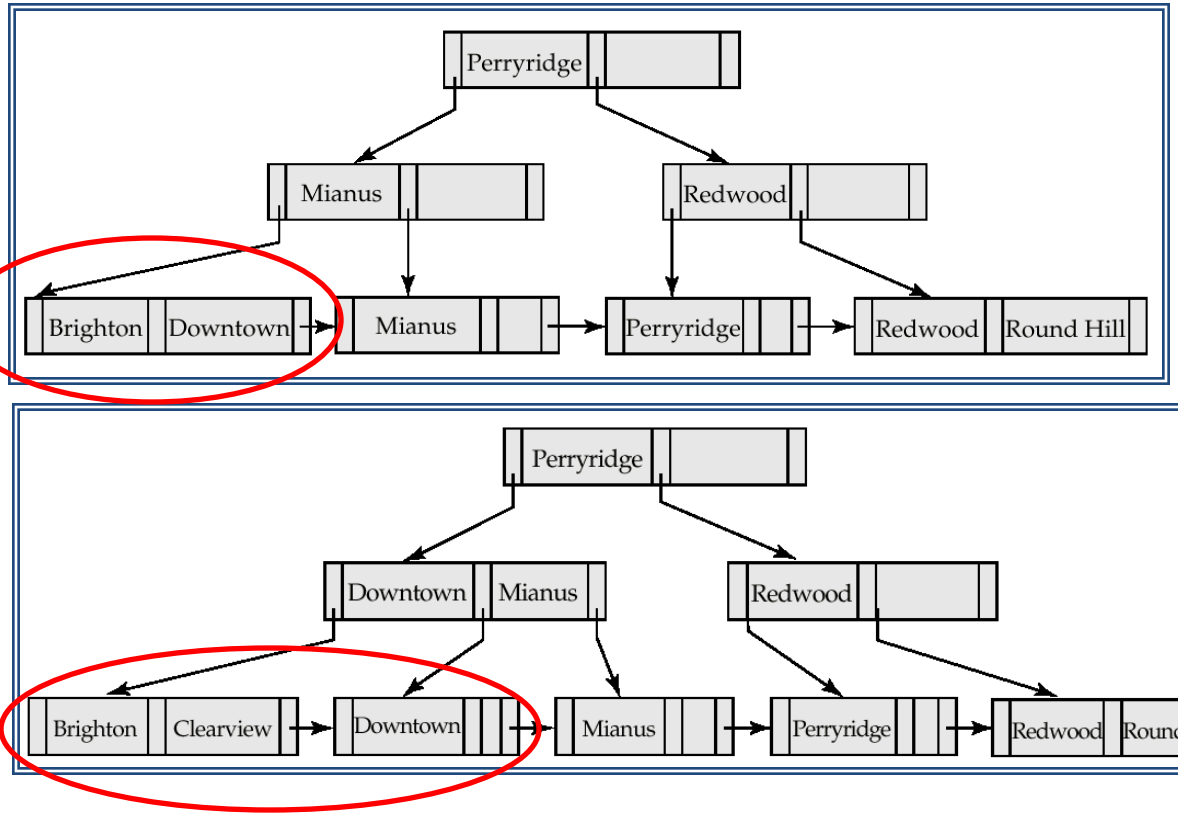
关于B + -树的查询(续)

- In processing a query, a path is traversed in the tree from the root to some leaf node
- If there are K search-key values in the file, **the path is no longer** than $\lceil log_{n/2}(K) \rceil$
  - E.g., a node is generally the **same size as a disk block**, typically **4 KB**, and n is typically around **100 (40 bytes per index entry)**
    - With **1** million search key values and n = **100**, at most $\log_{50}(1,000,000) = 4$ **nodes** are accessed in a lookup.
    - For a balanced binary tree with **1** million search key values — around **20 nodes** (i.e., $\log_2(1,000,000)$) are accessed in a lookup
    - The above difference is significant since every node access may need a disk I/O, costing around **10** ms

插入B + -Tree

- **找到搜索键值出现的叶节点**
  - 如果搜索键值已经在叶节点中
    - 记录被添加到文件中
    - 必要时，在桶中插入一个指针
  - 如果搜索键值不在某个节点，则将该记录添加到主文件中，并根据需要创建桶。然后:
    - 如果叶节点有空间，则在叶节点中插入(键值，指针)对
    - 否则，将该节点与新的(键值，指针)条目一起拆分

插入B + -树(续)



N = 3

插入"Clearview"前后的B+ -树

插入B + -树(续)

- **Splitting a leaf node**
  - take the n (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node
  - let the new node be $\boldsymbol{p}$, and let $\boldsymbol{k}$ be the least key value in $\boldsymbol{p}$. Insert (k, p) in the parent of the node being split
  - If the parent is full, split it and propagate the split further up
- **Splitting of nodes proceeds upwards till a node that is not full is found**
  - In the worst case, the root node may be split, thus increasing the height of the tree by **1**

插入B+-树(续)

- **Splitting a non-leaf node**: when inserting (k, p) into an full internal node N
  - Copy N to an in-memory area M with space for $n+1$ **pointers** and n **keys**
  - Insert (k, p) into M
  - Copy $P_1, K_1, \ldots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ from M back into node N
  - Copy $P_{\lceil n/2 \rceil + 1}, K_{\lceil n/2 \rceil + 1}, \ldots, K_n, P_{n+1}$ from M into the newly allocated node N'
  - Insert $(K_{\lceil n/2 \rceil}, N')$ into parent N

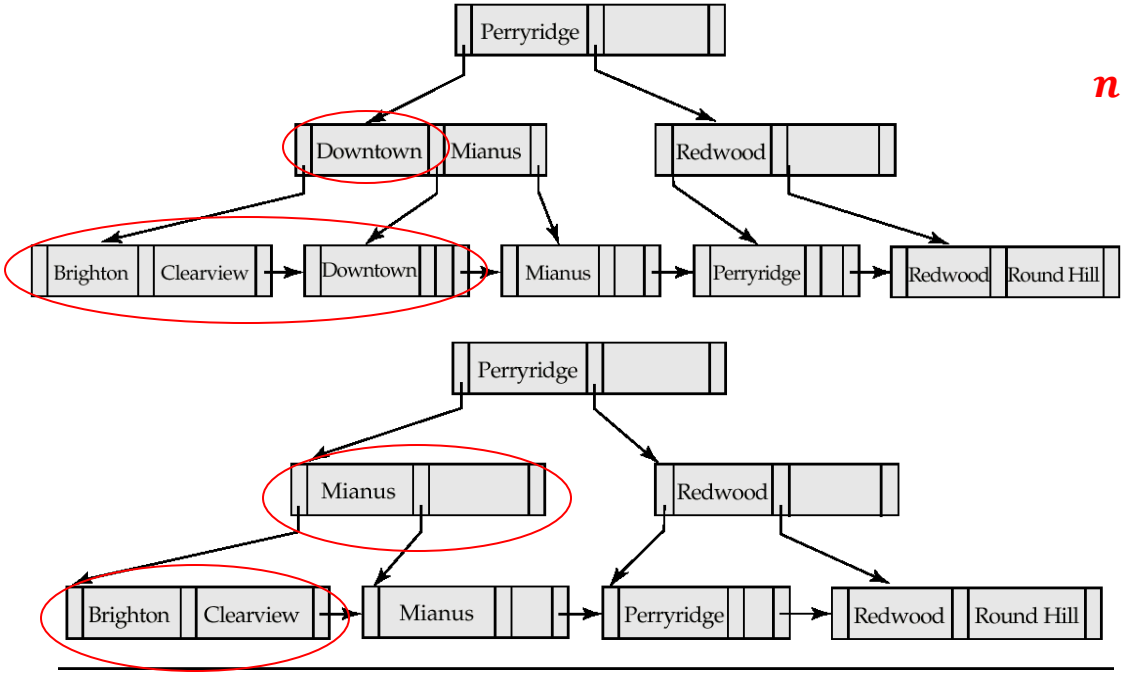
P_1	K_1	P_2	$\cdots$	P_{n-1}	K_{n-1}	P_n

B + -树中的删除

- Find the record to be deleted, and remove it from the main file and from the bucket
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then **merge siblings**
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node
  - Delete the pair (K_{i-1}, P_i), where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure

B + -树删除示例

删除
"Downtown"前
后对比



$n = 3$

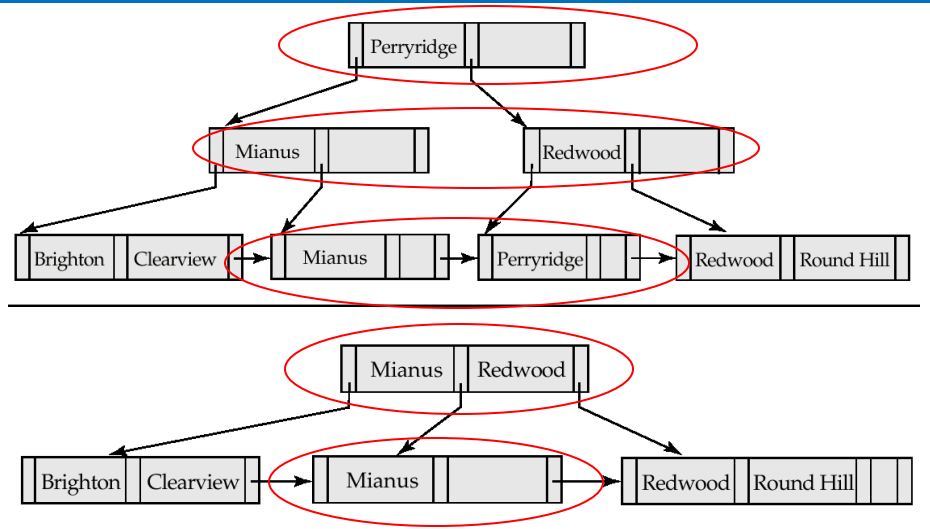
- 删除"Downtown"会导致欠满叶的合并
- 删除包含"Downtown"的叶节点不会导致其父节点指针过少。因此，级联删除在被删除的叶节点的父节点处停止

B + -树中的删除(续)

- If the node has too few entries due to the removal, and the entries in the node and a sibling don't fit into a single node, then **redistribute pointers**
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries
  - Update the corresponding search-key value in the parent of the node
- The node deletions may cascade upwards till a node which has $\lceil n/2 \rceil$ or more pointers is found.
- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root

B + -树删除示例(续)

删除"Perryridge"

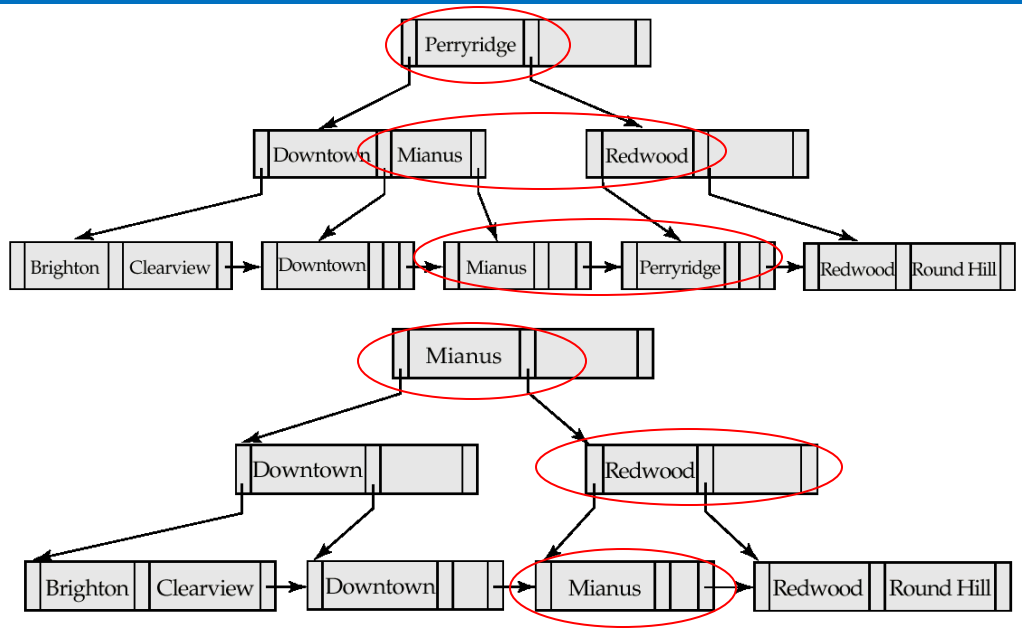


N = 3

- 具有"Perryridge"的节点变为未满(在这种特殊情况下实际上是空的)并与它的兄弟节点合并
- 因此，"Perryridge"节点的父节点变得不完整，并与其兄弟节点合并(并且从它们的父节点中删除了一个条目)。
- 根节点然后只有一个子节点，并且被删除，它的子节点成为新的根节点

B + -树删除示例(续)

删除"Perryridge"

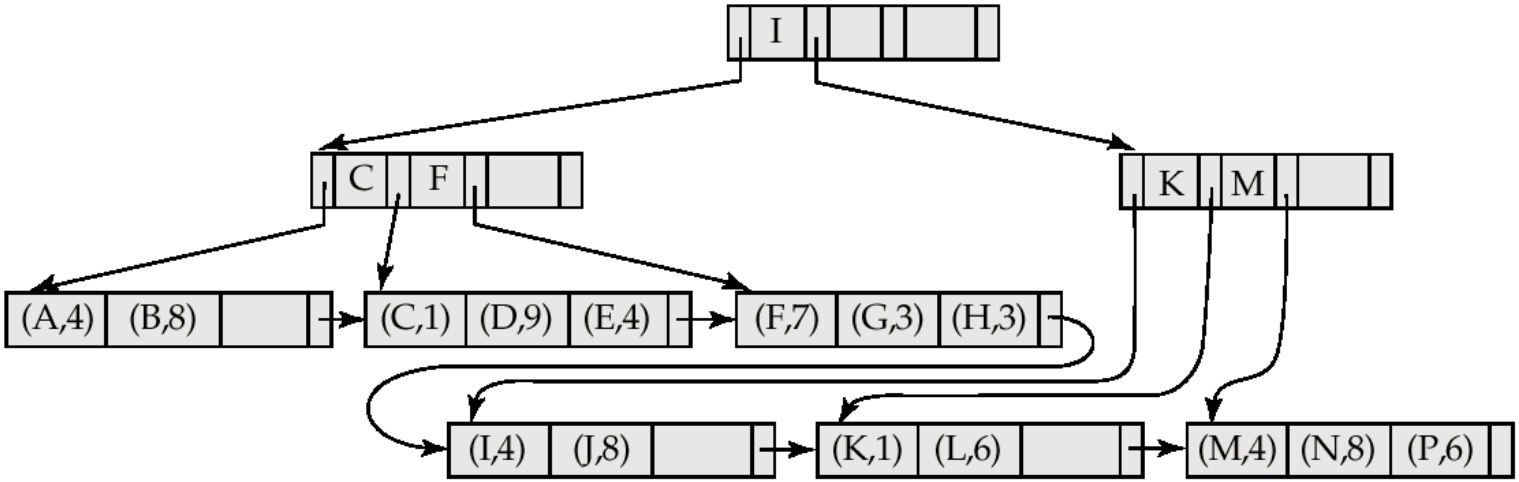


- 含有Perryridge的叶子的父结点变得欠满，并从它的左兄弟结点借用了一个指针
- 结果，父节点的父节点中的搜索键值发生了变化

B+ -树文件组织

- 通过使用B + -树索引解决了索引文件退化问题。采用B + -树文件组织(B +)解决数据文件降级问题。
- B +树文件组织中的叶节点存储记录，而不是指针
- 由于记录比指针大，因此叶子节点中可以存储的最大记录数小于非叶子节点中的指针数
- 叶节点仍然要求是半满的
- 插入和删除的处理方式与B +树索引中条目的插入和删除相同

B+ -树文件组织(续)

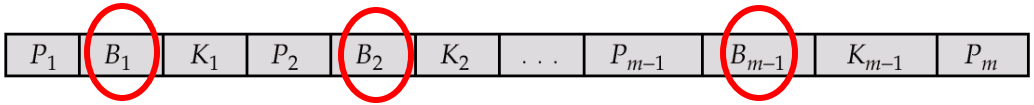


- 良好的空间利用率很重要，因为记录比指针使用更多的空间。
- 为了提高空间利用率，在重新分配中涉及更多的兄弟节点
  - 在重新分配中涉及2个或更多的兄弟节点，以尽可能避免分裂/合并

b树索引文件

- Similar to B+-tree, but **B-tree** allows search-key values to appear **only once,** thus eliminating redundant storage of search keys
- Search keys in non-leaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a non-leaf node is included
- **Generalized B-tree leaf node**

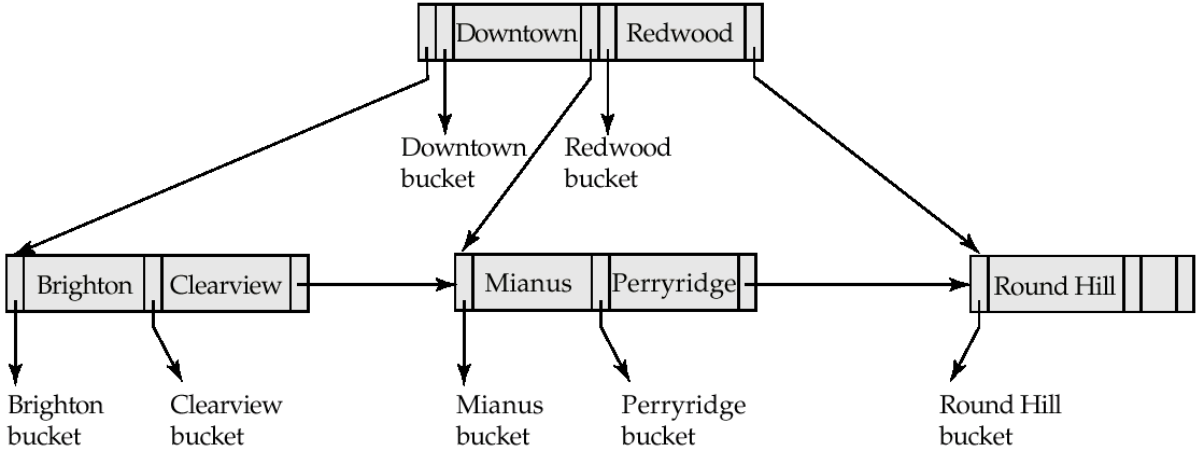
(a) 叶子节点



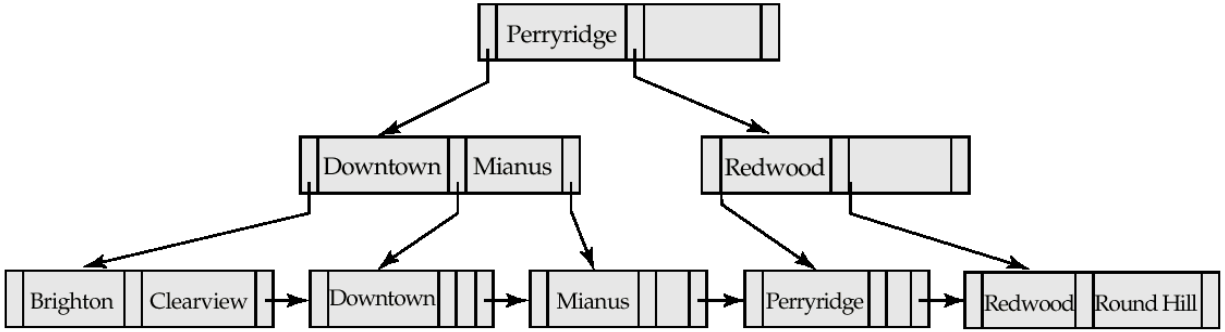
(b) 非叶节点

- **Nonleaf node** – pointers B_i are the bucket or file record pointers

b树索引文件



B-tree(上图)和B + -tree(下图)在同一数据上



b树索引文件(续)

- **b树索引的优点**
  - 使用比B + -树更少的树节点
  - 有时可能在到达叶节点之前找到搜索键值。
- **b树索引的缺点**
  - 只有一小部分的搜索键值能在早期被发现
  - 非叶节点更大，因此扇出减少。因此，B树通常比B + -树具有更大的深度
  - 插入和删除比B + -树更复杂
  - 实现比B + -树更难
- 通常，B树的优点不会超过缺点

大纲

- 基本概念
- 有序索引
- B+ -tree & B-tree索引

静态和动态哈希

- 有序索引与哈希
- SQL中的索引定义
- 多个键访问

静态哈希

- A bucket is a unit of storage containing one or more records (a bucket is typically a disk block)
- In a hash file organization, we obtain the bucket of a record directly from its search-key value using a hash function
- **Hash function h** is a function from the set of all search-key values K to the set of all bucket addresses B
- Hash function is used to locate records for access, insertion as well as deletion
- Records with different search-key values may be mapped to the same bucket; thus the entire bucket has to be searched sequentially to locate a record

哈希文件组织示例(续)

- 账户文件的哈希文件组织，使用branch-name作为键(见下一张图)
  - 一共有10个桶
    - 假定第i个字符的二进制表示为整数i
    - 哈希函数返回字符的二进制表示形式对10取模的和
    - 例如，h(Perryridge)= 125 mod 10 = 5小时(圆丘)= 113 mod 10 = 3小时(布莱顿)= 93 mod 10 = 3

哈希文件组织示例

帐户文件的哈希文件组织，使用branch-name作为键。

假定第i个字符的二进制表示为整数i

h(Perryridge)= 125 mod 10 = 5 h(圆丘)= 113 mod 10 = 3 h(布莱顿)= 93 mod 10 = 3

bucket 0

bucket 1

bucket 2

bucket 3

A-217	Brighton	750
A-305	Round Hill	350

bucket 4

A-222	Redwood	700

bucket 5

A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700

bucket 6

bucket 7

A-215	Mianus	700

bucket 8

A-101	Downtown	500
A-110	Downtown	600

bucket 9

哈希函数

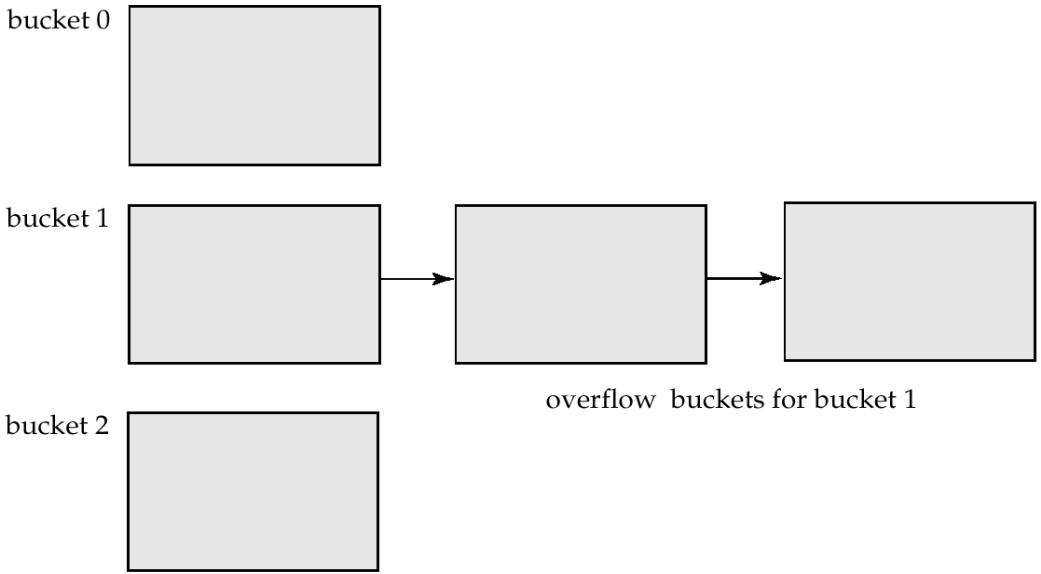
- 最差哈希函数将所有搜索键值映射到同一桶
- **理想的哈希函数是均匀的，即每个桶从所有可能值的集合中分配相同数量的搜索键值**
- **理想哈希函数是随机的，因此无论文件中搜索键值的实际分布如何，每个桶都会有相同数量的记录分配给它**
- **典型的哈希函数在搜索键的内部二进制表示上执行计算**

桶溢出的处理

- **桶溢出可能发生，因为**
  - 不足的桶
  - 记录分布不均匀。这种情况的发生有两个原因:
    - 多个记录具有相同的搜索键值
    - 选定的散列函数产生键值的非均匀分布
- 虽然可以降低桶溢出的概率，但无法消除;是通过使用溢出桶来处理的

桶溢出的处理(续)

- 溢出链-给定桶的溢出桶被链在一个链表中



散列索引

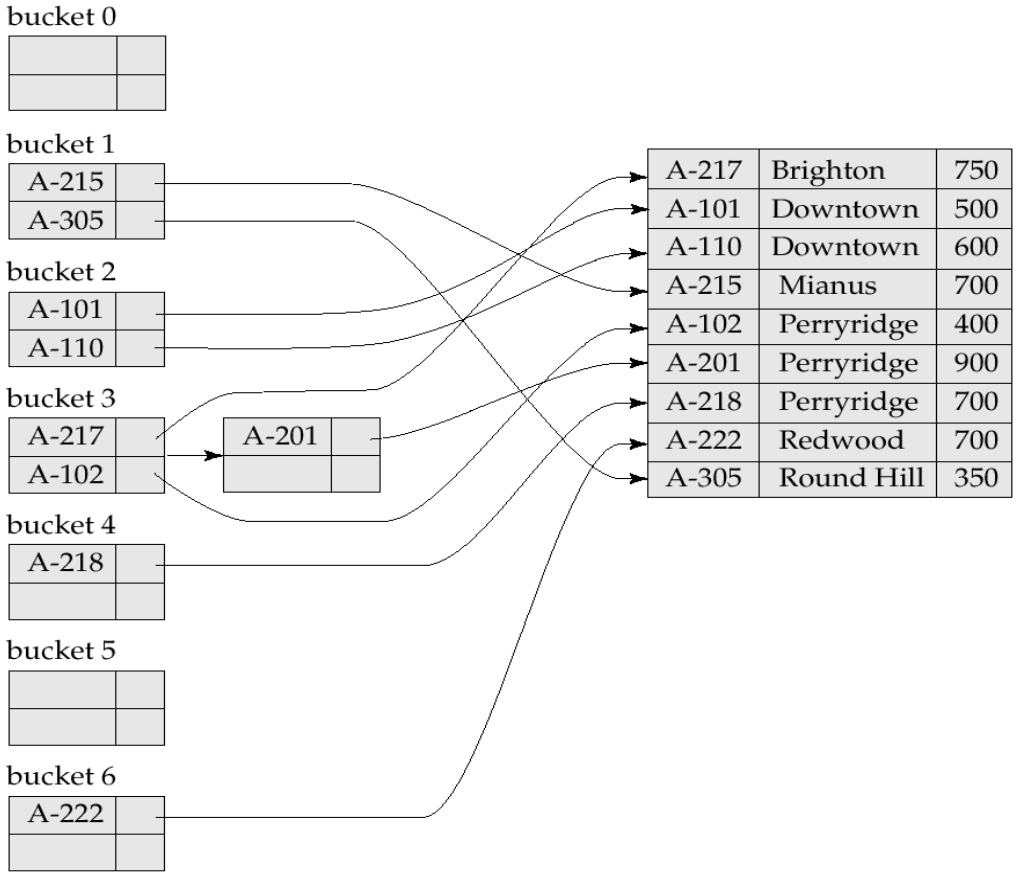
- **哈希不仅可以用于文件组织，还可以用于创建索引结构**
- **哈希索引将搜索键及其关联的记录指针组织成一个哈希文件结构**
- 严格地说，哈希索引始终是二级索引
  - 如果文件本身是使用哈希来组织的，那么使用相同的搜索键在其上建立一个单独的主哈希索引是不必要的
  - 然而，我们使用术语哈希索引来指代二级索引结构和哈希组织的文件

哈希索引示例

帐户文件上的二级哈希索引，用于搜索键account_number。

哈希函数计算账号的位数对7取模的和。

哈希索引有7个桶，每个桶的大小为2。一个有一个溢出桶。



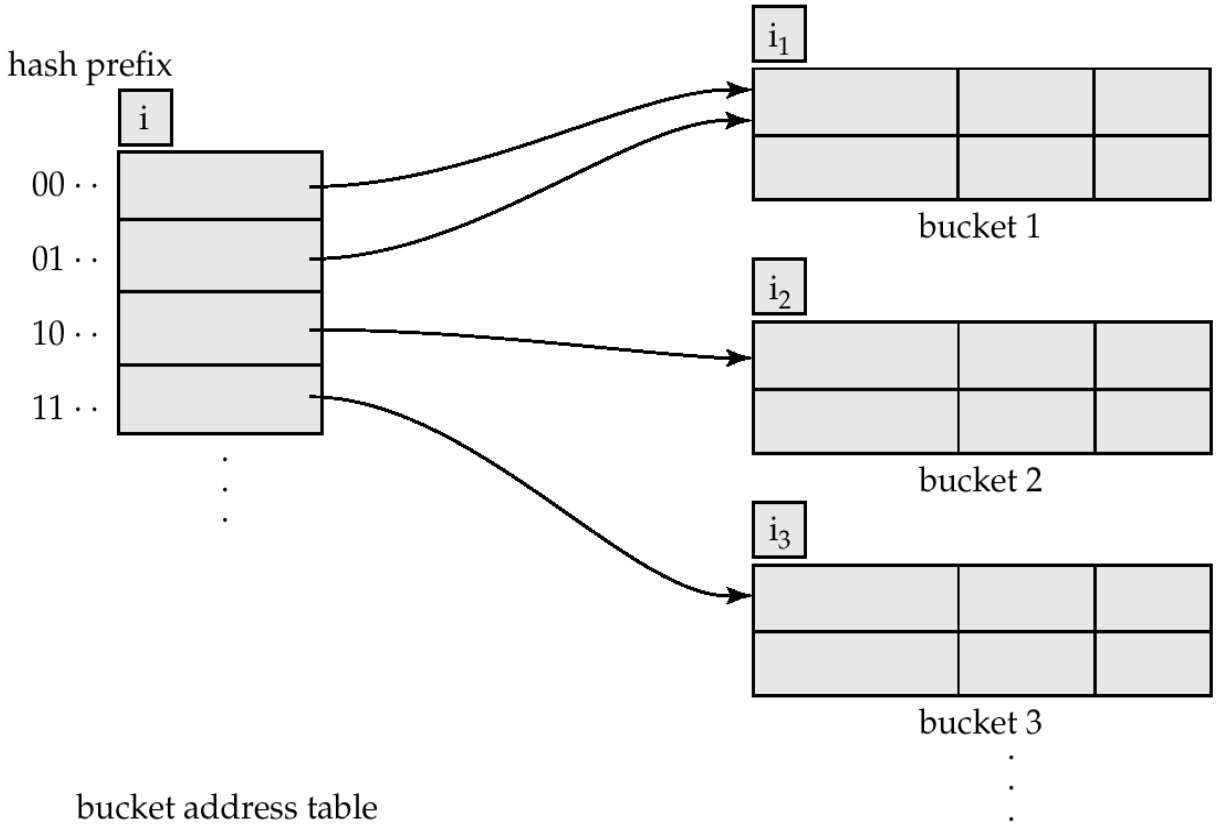
静态哈希的不足

- In **static hashing**, function h maps search-key values to a fixed set of B bucket addresses
  - Databases grow with time. If the initial number of buckets is too small, performance will degrade due to too much overflows
  - If file size at some point in the future is anticipated and choose the number of buckets allocated accordingly, significant amount of space will be wasted initially
  - If database shrinks, again space will be wasted
  - One option is periodic re-organization of the file with a new hash function, but it is very expensive.
- These problems can be avoided by using techniques that allow the number of buckets to be modified **dynamically**

动态哈希

- Good for database that grows and shrinks in size
  - Allows the hash function to be modified dynamically
  - **Extendable hashing(可扩充散列)** – one form of dynamic hashing
  - Hash function generates values over a large range - typically b-bit integers, with b = 32 (then 2^{32} hash values).
  - At any time use only a **prefix of the hash function** to index into a table of bucket addresses.
  - Let the length of the prefix be i **bits,** $0 \leq i \leq 32$
  - Bucket address table size = 2^i. Initially $i = 0$
  - Value of i grows and shrinks as the size of the database grows and shrinks.
  - Multiple entries in the bucket address table may point to a bucket
  - Thus, **actual number of buckets is** $< 2^i$
    - The number of buckets also changes dynamically due to coalescing and splitting of buckets.

通用可扩展哈希结构


hash prefix
i
00 · ·
01 · ·
10 · ·
11 · ·
i1
bucket 1
i2
bucket 2
i3
bucket 3
bucket address table

可扩展哈希结构的使用

- Each bucket j stores a value i_j; all the entries that point to the same bucket have the same values on the first i_j bits.
- To **locate** the bucket containing search-key K_j:
  - 1. Compute $h(K_j) = X$
  - 2. Use the **first i high order bits** of X as a displacement into bucket address table, and follow the pointer to appropriate bucket
- To **insert** a record with search-key value K_j
  - follow same procedure as look-up and locate the bucket, say j
  - If there is room in the bucket j insert record in the bucket.
  - Else the bucket must be split and insertion re-attempted (next slide.)
    - **Overflow buckets** used instead in some cases (will see shortly)

可扩展哈希结构的更新

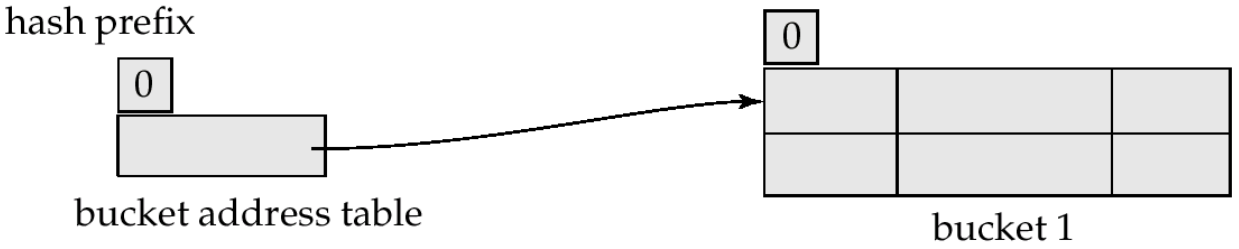
- To split a bucket j when **inserting record** with search-key value K_j:
  - **If** $i > i_j$ (more than one pointer to bucket j)
    - allocate a new bucket z, and set i_j and i_z to the old $i_j + 1$
    - make the second half of the bucket address table entries pointing to j to point to z
    - remove and reinsert each record in bucket j
    - recompute new bucket for K_j and insert record in the bucket (further splitting is required if the bucket is still full)
  - **If** $i = i_j$ (only one pointer to bucket j)
    - **increment** i and **double the size of the bucket address table**.
    - replace each entry in the table by two entries that point to the same bucket.
    - recompute new bucket address table entry for K_j
      Now $i > i_j$ so use the first case above.

可扩展散列结构的更新(续)

- **When inserting a value**, if the bucket is full after several splits (that is, *i* reaches some limit *b*) create an overflow bucket instead of splitting bucket entry table further.
- **To delete a key value**,
  - locate it in its bucket and remove it.
  - The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
  - Coalescing of buckets can be done (can coalesce only with a "buddy" bucket if it is present)
  - Decreasing bucket address table size is also possible
  - **Note:** decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table

可扩展散列结构的使用:示例

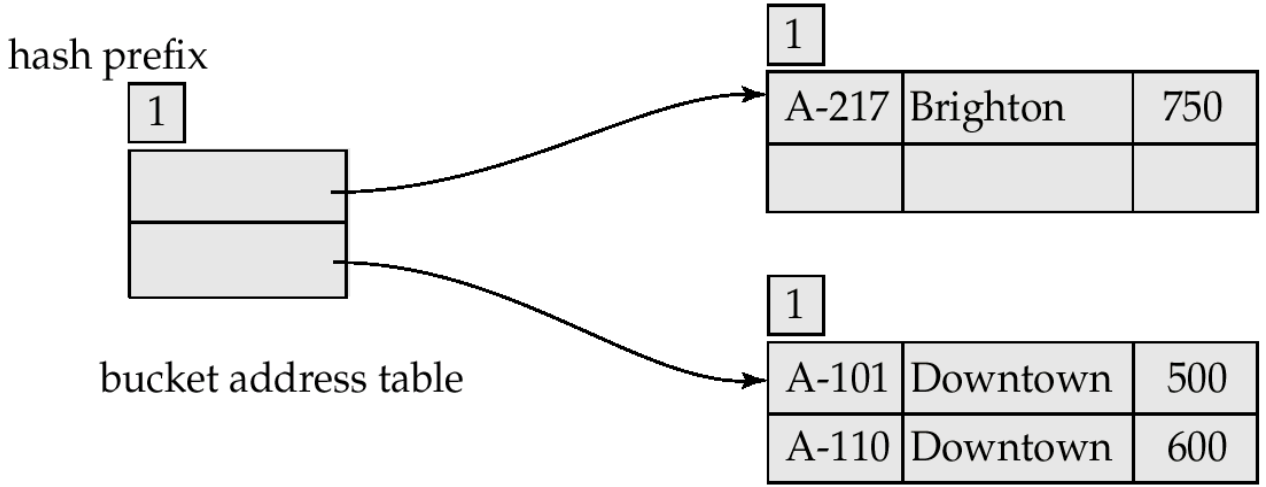
branch-name	h(*branch-name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001



初始哈希结构，桶大小= 2

例子(续)。

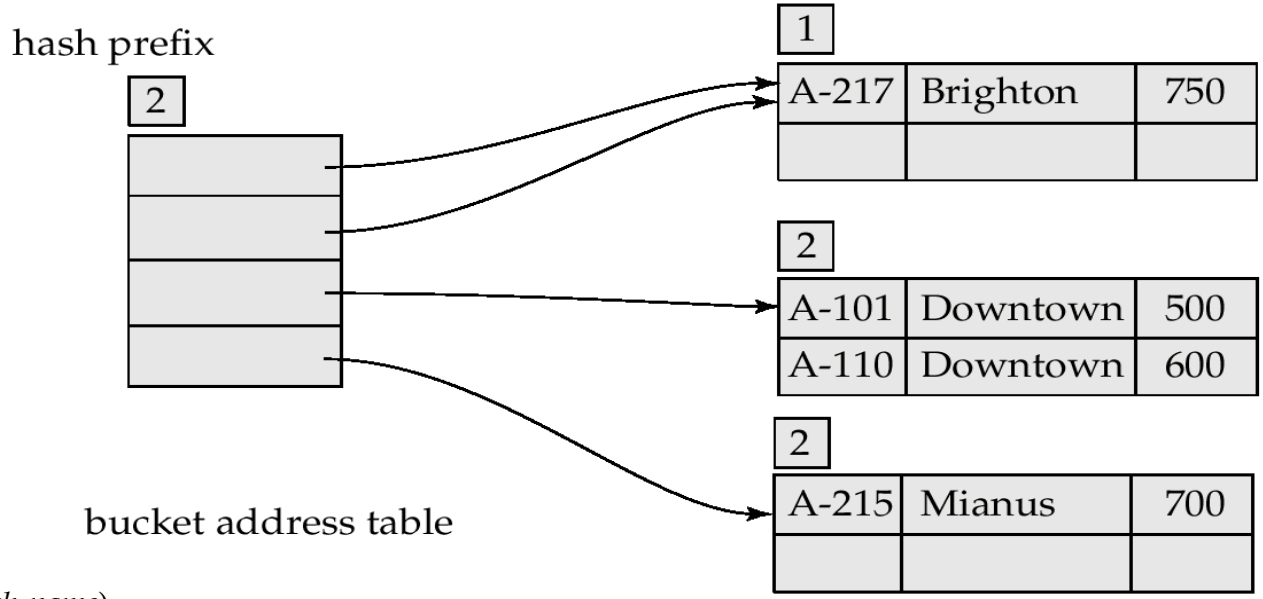
- 插入一条Brighton记录和两条Downtown记录后的哈希结构



branch-name	h(*branch-name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001

例子(续)。

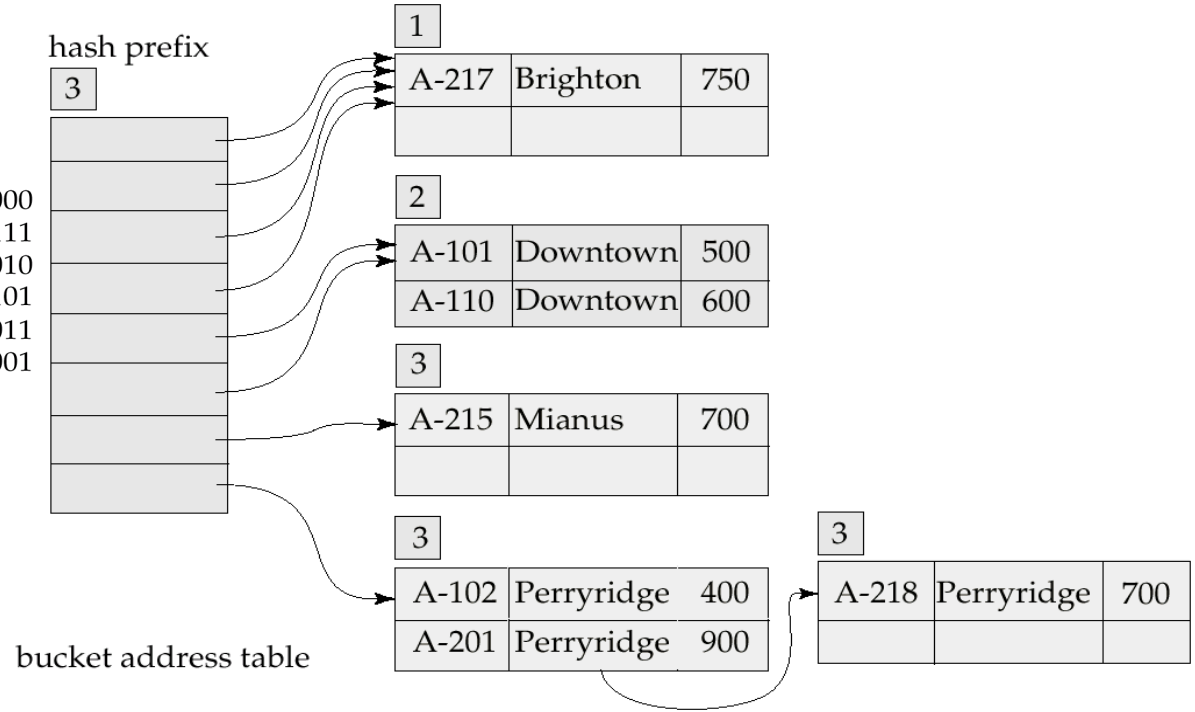
- 插入Mianus记录后的哈希结构



branch-name	h(*branch-name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001

例子(续)。

branch-name	h(*branch-name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001

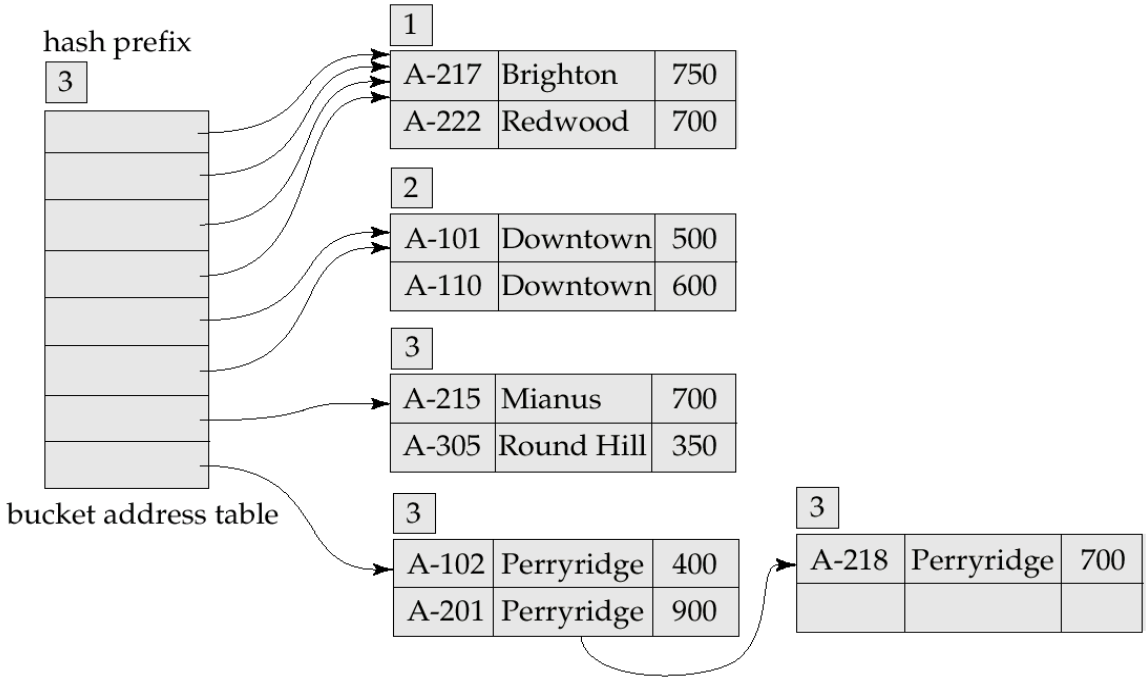


插入三条Perryridge记录后的哈希结构

例子(续)。

- 插入Redwood和Round Hill记录后的哈希结构

branch-name	h(*branch-name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001



可扩展哈希与其他方案

- 可扩展哈希的好处:
  - 哈希性能不会随着文件的增长而降低
  - 最小的空间开销
- 可扩展散列的缺点
  - 额外的间接层来找到想要的记录
  - 桶地址表本身可能会变得非常大(大于内存)
    - 需要树形结构来定位结构中想要的记录!
  - 改变桶地址表的大小是一个代价昂贵的操作
- 线性哈希是一种替代机制，它避免了这些缺点，但代价可能是更多的桶溢出

大纲

- 基本概念
- 有序索引
- B+ -tree & B-tree索引
- 静态和动态哈希

有序索引与哈希

- SQL中的索引定义
- 多个键访问

有序索引与哈希

- **文件组织和索引的问题**
  - 定期重新组织的成本
  - 插入和删除的频率
  - 是否以牺牲最坏情况访问时间为代价来优化平均访问时间
  - 预期的查询类型
    - 哈希通常更适合检索具有指定键值的记录
    - 如果范围查询是常见的，则首选有序索引

大纲

- 基本概念
- 有序索引
- B+ -tree & B-tree索引
- 静态和动态哈希
- 有序索引与哈希

SQL中的索引定义

- 多个键访问

SQL中的索引定义

- **创建索引**

  create [UNIQUE] index <index-name> on < relationship -name> (<attribute-list>)

  例如，在分支(branch_name)上创建索引b_index

  - 使用create unique index间接指定和强制搜索键是候选键的条件
    - 如果支持SQL唯一完整性约束，则不需要
- **删除索引**

  删除索引<index-name>

大纲

- 基本概念
- 有序索引
- B+ -tree & B-tree索引
- 静态和动态哈希
- 有序索引与哈希
- SQL中的索引定义

→多键访问

多个键访问

- 对特定类型的查询使用多个索引
  - 例如，

选择account_number
从账户
where branch_name = " Perryridge " and balance = 1000

- **在单个属性上使用索引处理查询的三种可能策略**
  - 在branch_name上使用索引查找branch_name = " Perryridge "的账户，测试余额为$1000;。
  - 使用index on balance查找余额为$1000的账户;test branch_name = " Perryridge "。
  - 使用branch_name索引查找指向Perryridge分支的所有记录的指针。类似地，使用index on balance。取所得到的两组指针的交集

多个属性上的索引

- 假设我们在组合搜索键(branch_name，balance)上有一个索引
- 对于where子句，其中branch_name = " Perryridge "且balance = 1000，则组合搜索键上的索引将只获取满足这两个条件的记录

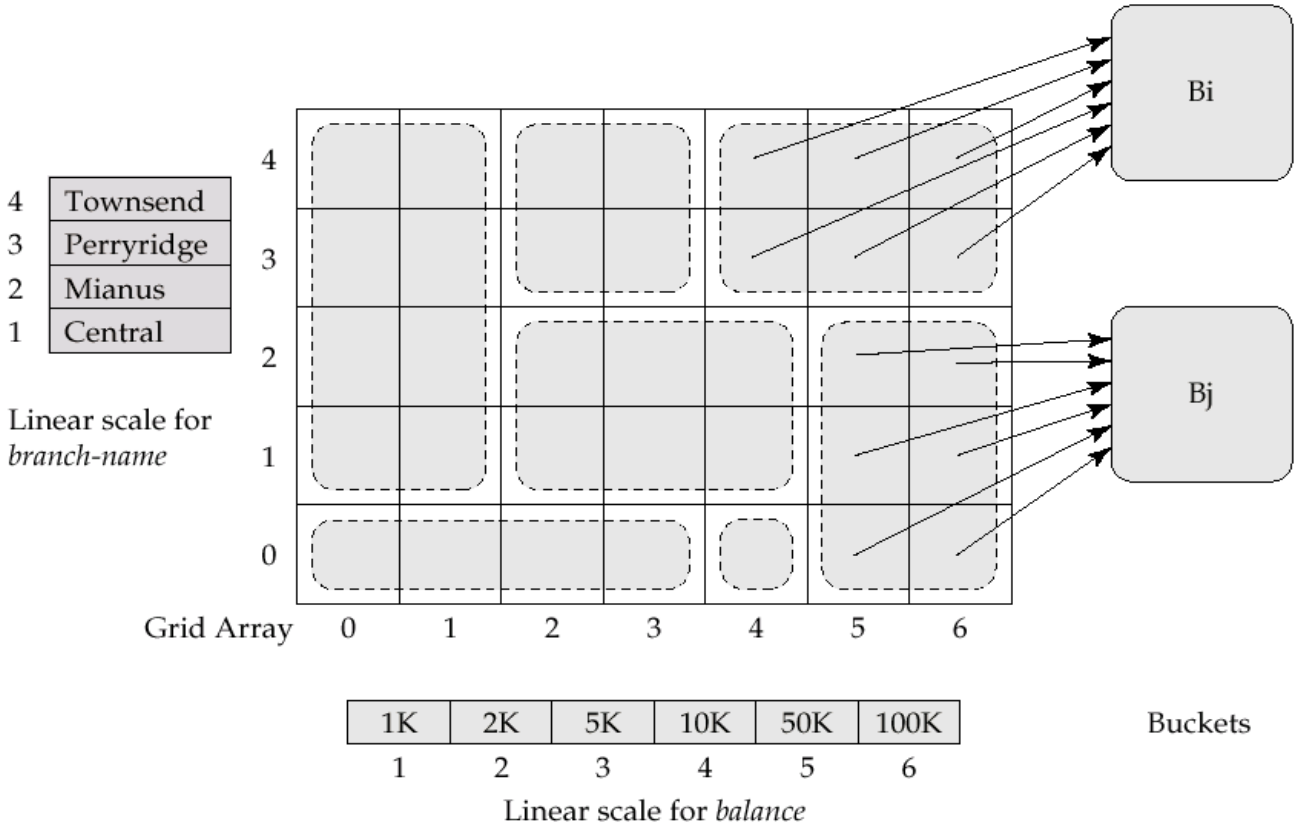
- 还可以有效地处理branch_name = " Perryridge "和balance < 1000

- 但是不能有效地处理branch-name < " Perryridge "和balance = 1000可能会获取许多满足第一个条件但不满足第二个条件的记录，可能导致大量I/ o

网格文件

- 用于加速涉及一个或多个比较运算符的一般多个搜索关键字查询的处理的结构
- **网格文件具有单个网格数组和每个搜索键属性的一个线性刻度。网格数组的维数等于搜索键属性的个数**
- 网格数组的多个单元格可以指向同一个桶
- 要找到搜索键值的桶，使用线性缩放和跟随指针定位其单元格的行和列

使用实例帐户网格文件


4 Townsend
3 Perryridge
2 Mianus
1 Central
Linear scale for branch-name
4
3
2
1
0
Grid Array 0 1 2 3 4 5 6
Bi
Bj
1K 2K 5K 10K 50K 100K
Buckets
1 2 3 4 5 6
Linear scale for balance

查询网格文件

- A grid file on two attributes ***A*** and ***B*** can handle queries of all following forms with high efficiency
  - $(a_1 \leq A \leq a_2)$
  - $(b_1 \leq B \leq b_2)$
  - $(a_1 \leq A \leq a_2 \wedge b_1 \leq B \leq b_2)$
- E.g.,
  - to answer $(a_1 \leq A \leq a_2 \wedge b_1 \leq B \leq b_2)$, use linear scales to find the corresponding candidate grid array cells, and look up all the buckets pointed to from those cells

网格文件(续)

- 在插入过程中，如果一个桶满了，如果有多个单元格指向它，就可以创建新的桶
  - 思路类似于可扩展哈希，只不过是在多个维度上
  - 如果只有一个单元格指向它，要么必须创建溢出桶，要么必须增加网格大小
- 必须选择线性尺度，以便在单元格之间均匀分布记录。
  - 否则会有太多溢出桶。
- 定期重新组织以增加网格大小会有所帮助
  - 但重组的成本可能非常高。
- 网格阵列的空间开销可能很高。

位图索引

- **Bitmap indices** are a special type of index designed for efficient querying on multiple keys
- Records in a relation are assumed to be numbered sequentially from:
  - Given a number n, it must be easy to retrieve record n
    - Particularly easy if records are of fixed size
- Applicable on attributes that take on a relatively small number of distinct values
  - E.g., gender, country, state, …
  - E.g., income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity)
- A bitmap is simply an array of bits

位图索引(续)

- In its simplest form, **a bitmap index** on an attribute has a bitmap for each value of the attribute
  - **Bitmap has as many bits as records**
  - In a bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise

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	10010
f	01101

Bitmaps for *income-level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Bitmap Indices (Cont.)

- **位图索引对于多个属性的查询很有用**
  - 对于单属性查询不是特别有用
- 查询使用位图操作来回答
  - **十字路口(和)**
  - **联盟(或)**
  - **互补(不)**
- 每个操作取两个大小相同的位图，并在对应的位上应用该操作，得到结果位图
  - 例如，100110 AND 110011 = 100010

    100110或110011 = 110111NOT100110 = 011001

  - 收入水平L1的男性:10010 AND 10100 = 10000
    - 然后可以检索所需的元组吗
    - 计算匹配元组的个数就更快了

位图索引(Cont.)

- 与关系大小相比，位图索引通常非常小
  - 例如，如果记录是100字节，单个位图的空间是关系使用空间的1/800。
    - 如果不同属性值的数量为8，则位图仅占关系大小的1%
- 删除需要妥善处理
  - 存在位图要注意某个记录位置是否存在有效记录
  - 需要进行补充
    - not(A=v):(NOT bitmap-A-v) AND ExistenceBitmap
- 应该为所有值保留位图，甚至是空值
  - 正确处理NOT(A=v)的SQL空语义:
    - 与(NOT bitmap-A-Null)相交

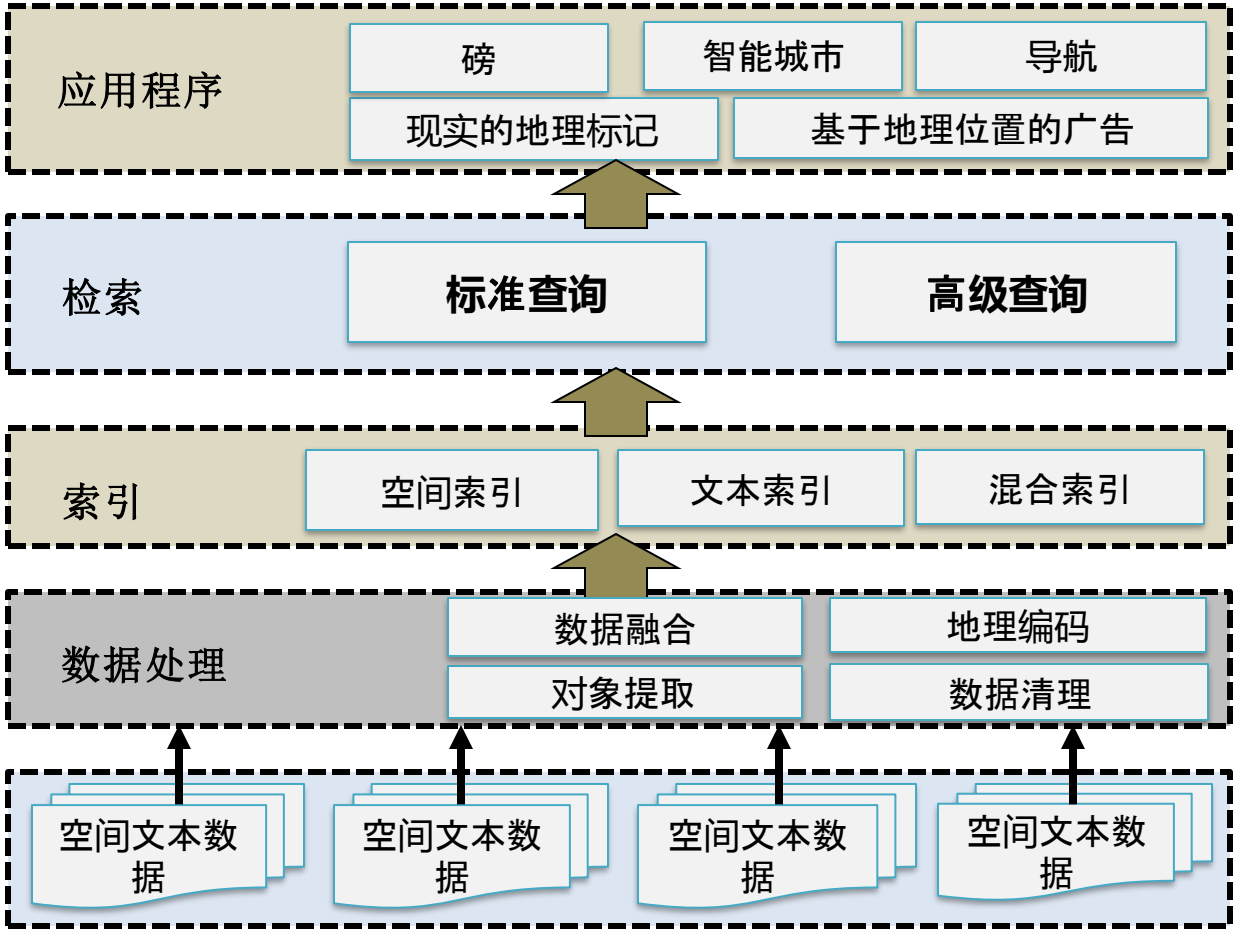
Assignments-Quiz

- **Q1:从一个空树构造一个B +树。每个节点可以存放4个指针**
  - 要插入的顺序值是:10,7,12,5,9,15,30,23,17,26
  - 然后分别删除9、10、15
  - 请给出每次插入和每次删除后的B +树
- **Q2:比较B +树和B-树，并描述它们的区别**

补充学习（索引相关）

- 商用数据库
  - Oracle: B,
    - “Oracle”,
  - IBM db2: b +
  - Microsoft SQL Server SQL Server
- 开源数据库
  - MySQL: B-Tree(B+Tree)， Hash
  - Postgre SQL, MySQL, Ingres r3, MaxDB, Firebird (InterBase)， MongoDB, SQLite, CUBRID, Cayley(图)
- NoSQL
  - HBase、Cassandra、MongoDB、Redis
  - OceanBase、openGauss人大金仓,X-DB,达梦……

研究框架


应用程序
磅
智能城市
导航
现实的地理标记
基于地理位置的广告
检索
标准查询
高级查询
索引
空间索引
文本索引
混合索引
数据处理
数据融合
地理编码
对象提取
数据清理
空间文本数据
空间文本数据
空间文本数据
空间文本数据

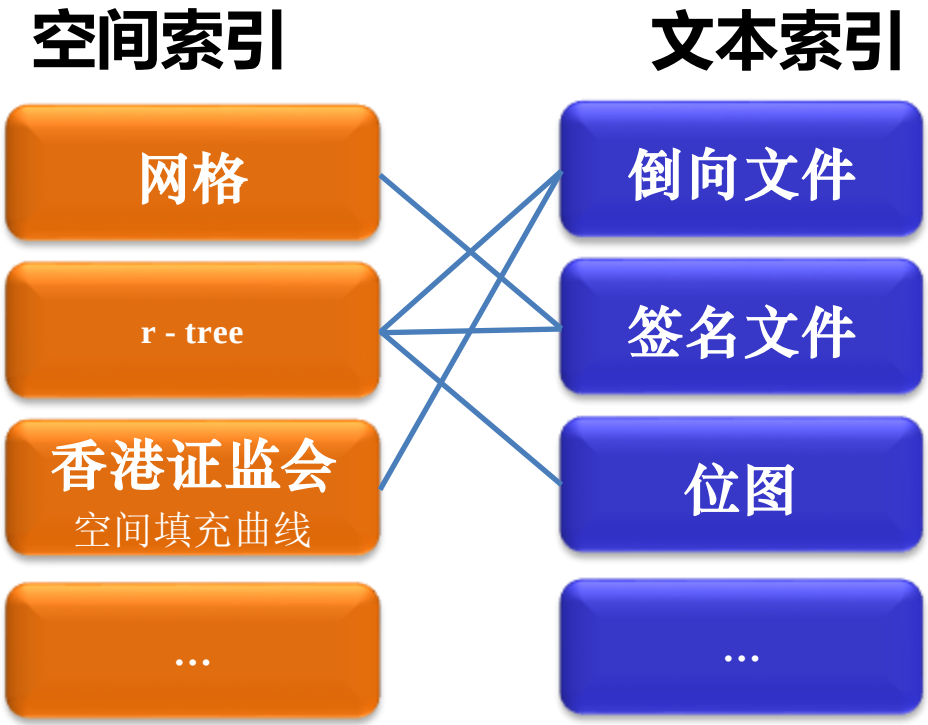
空间文本对象(spatial -textual objects)

- $o=(ld)$
  - $o。l$:空间定位，$o。d$:文字描述

POI:商店、银行、餐厅、博物馆、学校、医院等

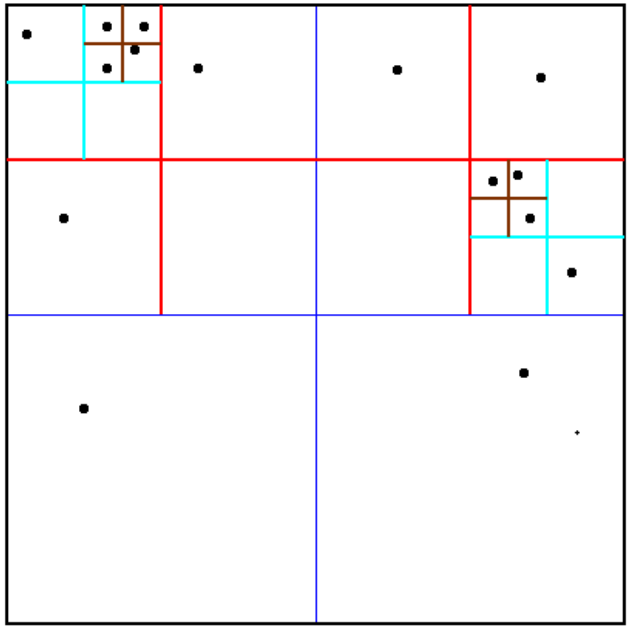
地理标记的网络内容:新闻、图片、视频、评论、微博

空间文本索引

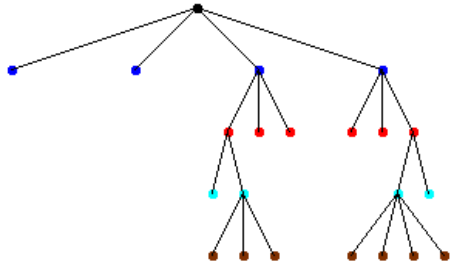


空间索引:网格索引

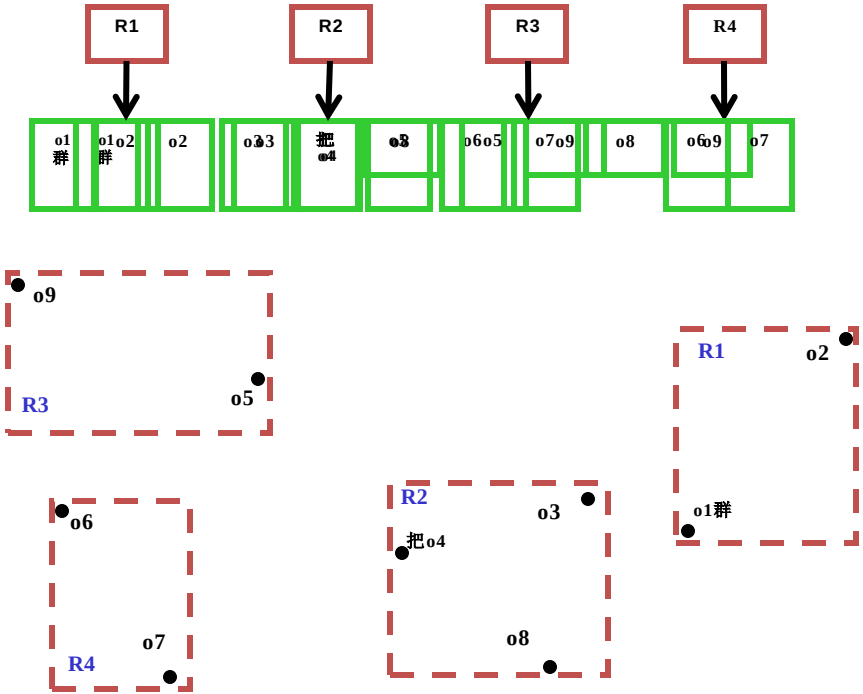
分区



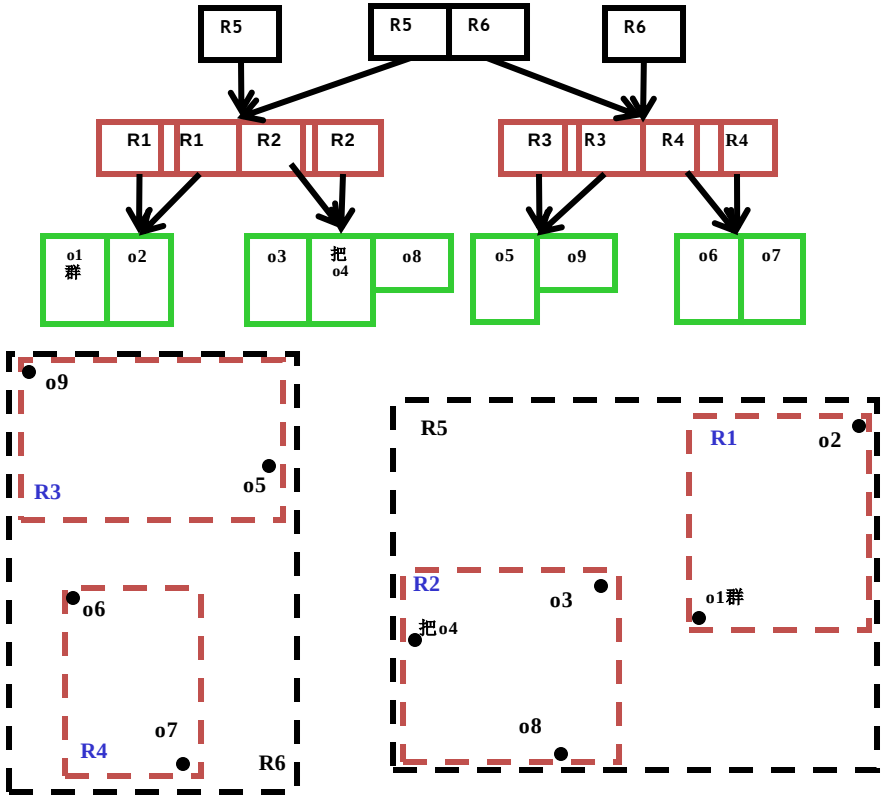
树状结构



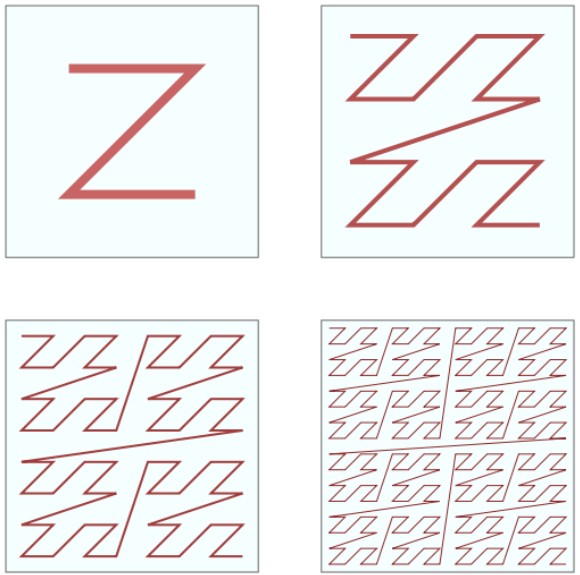
空间索引:r树


R1
R2
R3
R4
o1群
o1群
o2
o2
o3
o3
把
o4
o6
o5
o7
o9
o8
o6
o9
o7
o9
R3
o5
o6
R4
o7
R2
o3
把o4
o8
R1
o2
o1群

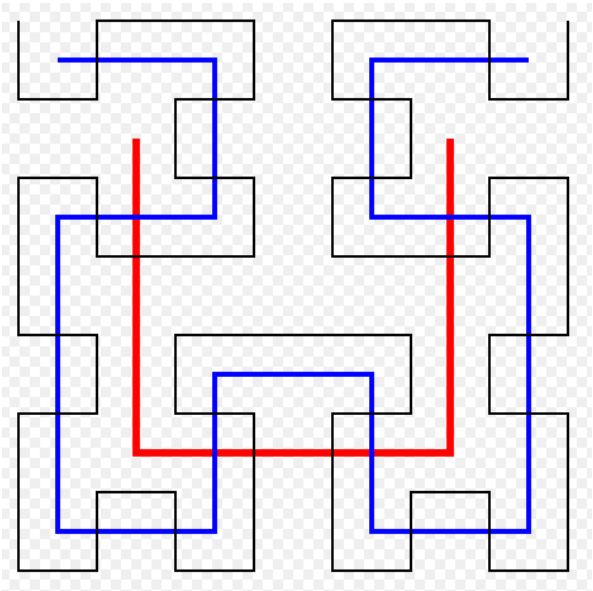
空间索引:r树


R5
R5
R6
R6
R1
R1
R2
R2
R3
R3
R4
R4
o1
群
o2
o3
把
o4
o8
o5
o9
o6
o7
o9
R3
o5
o6
R4
o7
R6
R5
R1
o2
R2
o3
o1群
把o4
o8

空间指数:空间填充曲线(SFC)

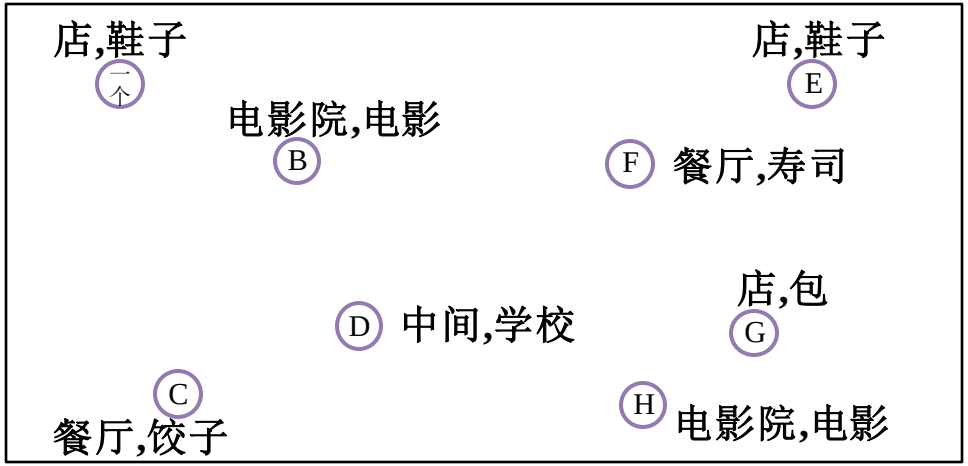


z曲线



希尔伯特曲线

文本索引:倒排索引



空间文本对象

关键字	Spatial-textual对象
商店	A, e, g
鞋子	A, E
电影	B、H
电影	B、H
餐厅	C、F
...	...

反向索引

文本索引:位图

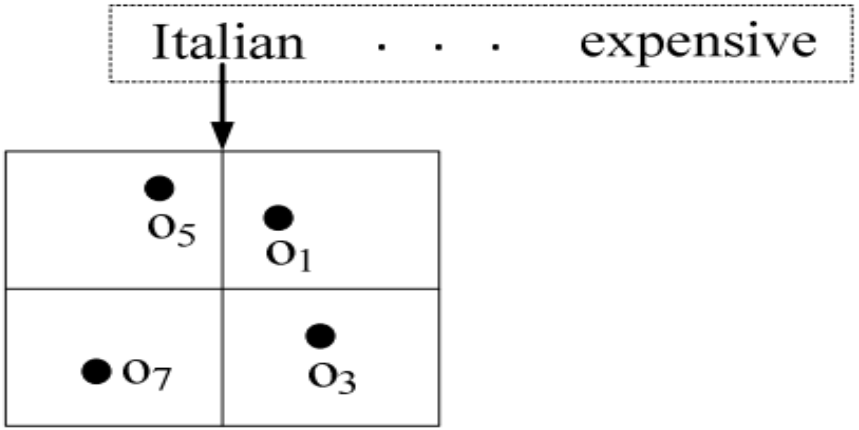
	k_1	k_2	k_3	k_4	k_5
$k_1 k_2 k_3$	1	1	1	0	0
$k_2 k_4 k_5$	0	1	0	1	1
$k_2 k_4$	0	1	0	1	0
$k_1 k_2 k_4 k_5$	1	1	0	1	1
$k_4 k_5$	0	0	0	1	1
…					

文本索引:签名文件

Terms/documents	Signature
k_1	$sig(k_1)$=0000000001
k_2	$sig(k_2)$=0000000010
k_3	$sig(k_3)$=1000000011
k_1k_2	$sig(k_1k_2)=sig(k_1) \vee sig(k_2)$=0000000011
k_2k_3	$sig(k_2k_3)=sig(k_2) \vee sig(k_3)$=1000000011
…	

ST指数& TS指数

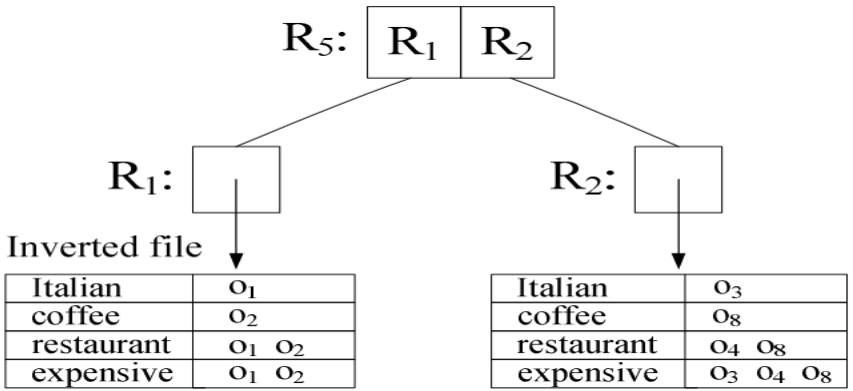
- 网格索引+倒排文件
  - ST:空间文本索引(网格索引优先)
  - TS:文本空间索引(倒排文件优先)



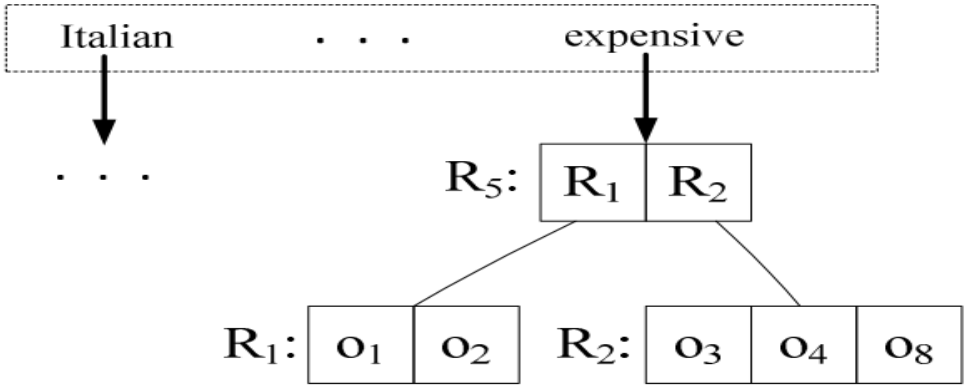
R*-tree- if和IF-R*-tree

- **R*-tree +倒立文件**
  - R*树:r树的一种变体

R *如果



如果r *树



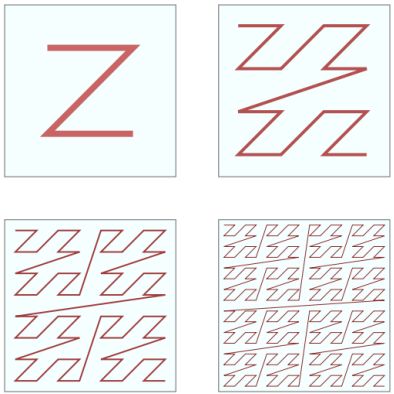
KR*-tree(关键词R*-tree)

- **R*-tree +倒排文件**
  - 每个节点实际上都增加了出现在其子树中的关键字集。
  - 节点被组织成倒排文件

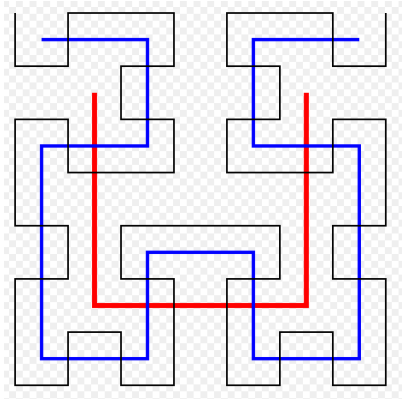
关键字	树节点
意大利	*R1, r 2, r 3, r 4, r 5, r 6*
咖啡	*R1, r2, r4, r5, r6*
餐厅	*R1, r2, r3, r4, r5, r6*
披萨	*R2, r4, r5, r6*
昂贵的	*R1 r 2 r 5*

证监会四边形

- **倒档+填充曲线**
  - **倒排文件+希尔伯特曲线:倒排列表在磁盘上沿希尔伯特曲线排列。**
  - **倒排文件+ z曲线:每个倒排列表中的对象根据其在z曲线上的空间位置进行分配和排序。**



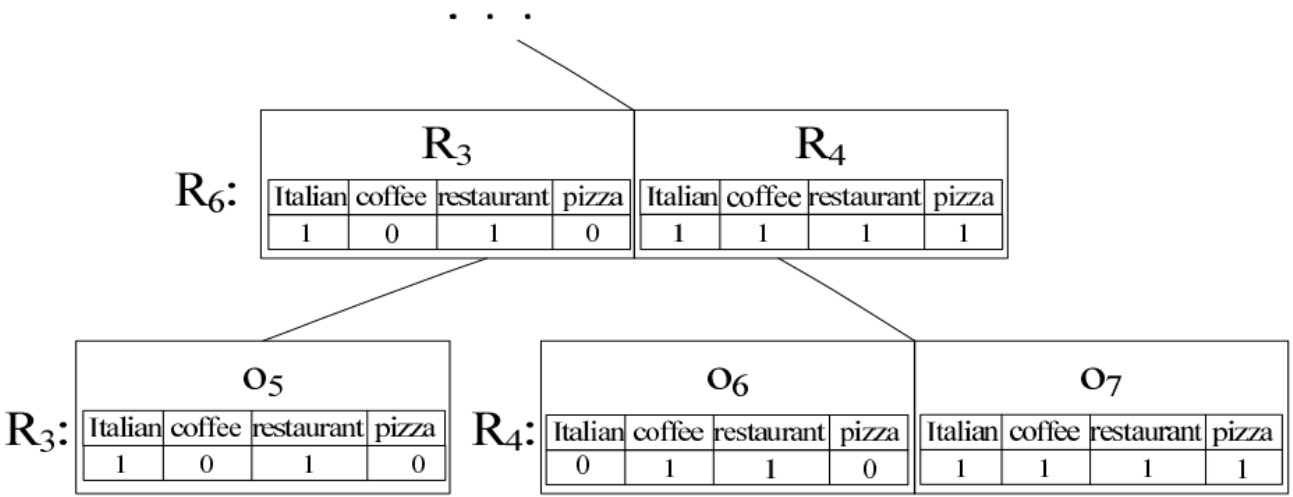
z曲线



希尔伯特曲线

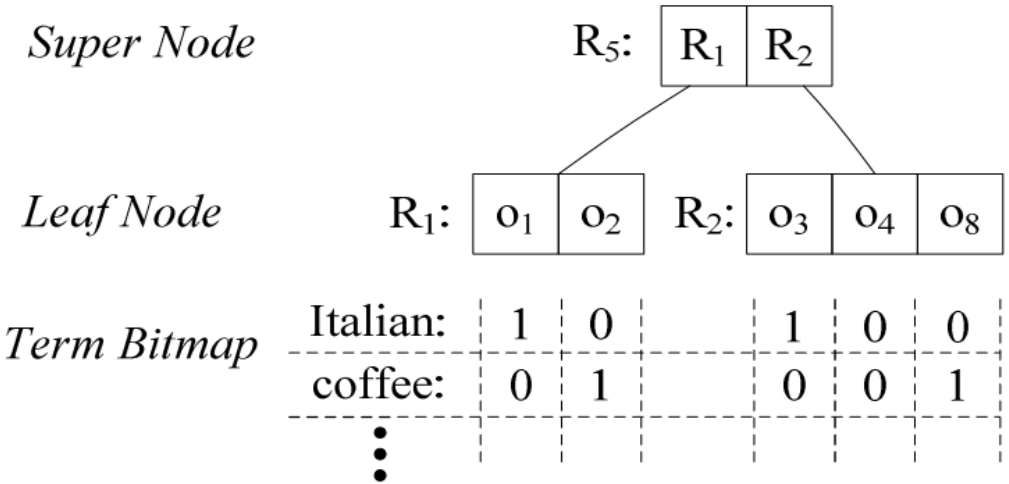
IR 2树

- 签名+ r树


. . .
R6:
R3
Italian coffee restaurant pizza
1 0 1 0
R4
Italian coffee restaurant pizza
1 1 1 1
R3:
o5
Italian coffee restaurant pizza
1 0 1 0
R4:
o6
Italian coffee restaurant pizza
0 1 1 0
o7
Italian coffee restaurant pizza
1 1 1 1

SKI (Spatial-Keyword Indexing)

- 位图+ r树

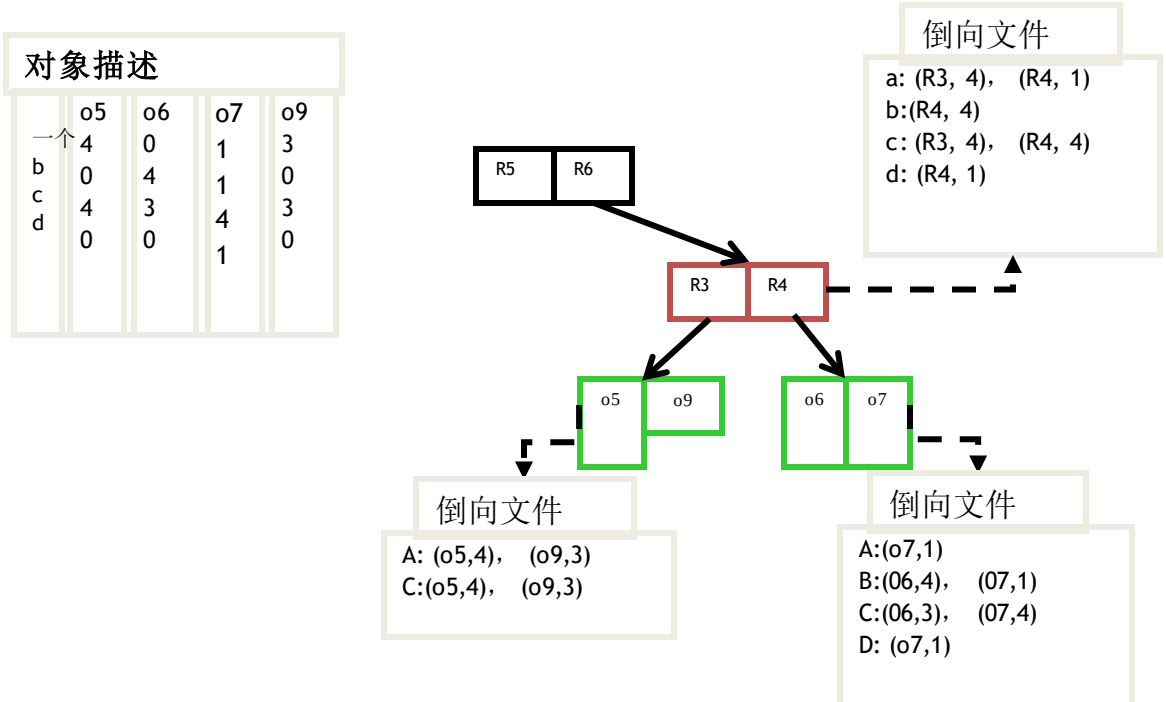


WIR-tree

- **r树+倒位图**
  - IR-tree的变体
- **的想法**
  - 考虑单词的频率
  - 按关键字频率递归地划分对象

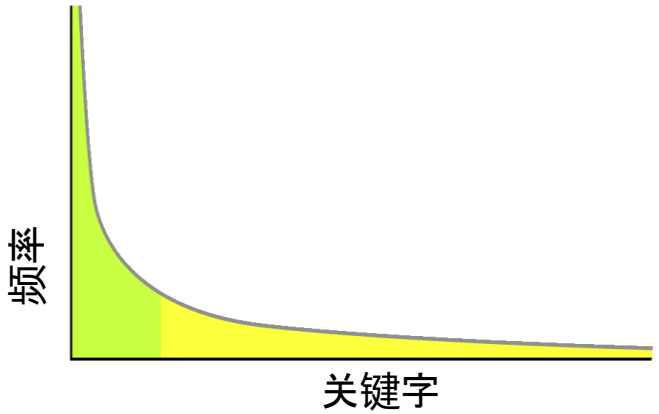
IR-tree

- 用子树中对象的文本内容摘要扩充r树的每个节点



S2I空间倒排索引

- **关键词歪斜分布**



- **S2I：r树+反向文件**
  - 先建倒排索引
  - 构建词频感知的空间索引
    - **常用关键词：聚合r树(aR树)**
    - **频率较低的关键词：块**

欧几里得空间中的SKQs

- **标准SKQs**
  - 布尔范围查询(BRQ)
    - 圣,TS
    - R *如果,如果R *树
    - 基米-雷克南*树
    - SKIF
  - 布尔kNN查询(BkQ)
    - IR2-tree
    - 滑雪
    - WIR-tree
  - Top-k查询(TkQ)
    - IR-tree
    - S2I
- **高级SKQs**
  - m-CK查询
  - 反向查询
  - 移动查询
  - 组查询
  - Direction-aware查询
  - 感兴趣区域查询
  - 为什么不能查询
  - 相似性连接查询
  - …

欧几里得空间中SKQ的指标

指数	空间索引	文本索引	结合	BkQ	TkQ	BRQ
圣	网格	如果	Spatial-first			√
TS	网格	如果	文本首先			√
如果r *树	R *树	如果	文本首先	Δ		√
R *如果	R *树	如果	Spatial-first		Δ	√
SF2I	香港证监会	如果	Spatial-first			√
基米-雷克南*树	R *树	如果	紧密结合	Δ		√
IR 2 -Tree	r - tree	位图	紧密结合	√		Δ
IR-Tree	r - tree	如果	紧密结合	Δ	√	Δ
SKIF	网格	如果	紧密结合			√
滑雪	r - tree	位图	Spatial-first	√		
S2I	r - tree	如果	文本首先	Δ	√	Δ
WIR-Tree	r - tree	发票。位图	紧密结合	√		Δ
SFC-QUAD	香港证监会	如果	紧密结合			√

总结

<s:1>基本概念

→有序索引

→B + -tree & B-tree索引

- 静态和动态哈希
- 有序索引与哈希
- **SQL中的索引定义**
- 多个键访问

第14讲结束