索引
Indexing

李文根/Wengen Li
Email: lwengen@tongji.edu.cn

先进数据与机器智能系统实验室（ADMIS Lab）
https://admis-tongji.github.io

同济大学 计算机科学与技术学院

2026年04月

课程概要

同济大学 TONGJI UNIVERSITY

- **Part 0: Overview**
  - Ch1: Introduction
- **Part 1 Relational Languages**
  - Ch2: Relational model
  - Ch3: Introduction to SQL
  - Ch4: Intermediate SQL
  - Ch5: Advanced SQL
- **Part 2 Database Design**
  - Ch6: Database design via E-R model
  - Ch7: Relational database design
- **Part 3 Application Design & Development**
  - Ch8: Complex data types
  - Ch9: Application development
- **Part 4 Big Data Analytics**
  - Ch10: Big data
  - Ch11: Data analytics
- **Part 5 Storage Management & Indexing**
  - Ch12: Physical storage systems
  - Ch13: Data storage structures
  - **Ch14: Indexing**
- **Part 6 Query Processing & Optimization**
  - Ch15: Query processing
  - Ch16: Query optimization
- **Part 7 Transaction Management**
  - Ch17: Transactions
  - Ch18: Concurrency control
  - Ch19: Recovery system
- **Part 8 Parallel & Distributed Database**
  - Ch20: Database system architecture
  - Ch21-23: Parallel & distributed storage, query processing & transaction processing
- **Advanced topics**
  - DB Platform: **OceanBase**, MongoDB, Neo4J
  - RAG, Multimodal retrieval, …

目录

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，访问速度快**
  - 如果数据表中一条记录在磁盘上占用1000B，对其中10B的一个字段建立索引，则该记录对应索引项的大小约为10B。如SQL Server的最小空间分配单元是页(Page)，一个页在磁盘上占用8KB空间，可以存储**8条**上述记录，可以存储索引项**800条**
  - 从一个有8000条记录的表中检索符合某条件的记录：
    - 如无索引，可能需要遍历8000条×1000B/8KB=**1000个页面**才能找到结果
    - 如果检索字段有上述索引，则可以在8000条×10B/8KB=**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I/O访问量要少很多

基本概念

- **Indexing mechanisms**
  - speed up the access to desired data
  - index files are typically much smaller than the original data file
- **Search Key(搜索码/关键字)**
  - the set of attributes used to look up records in a file/table
  - an index file consists of records (called index entries, 索引项) of the form (search-key, pointer)
- **Two types of indices**
  - **ordered index (顺序索引):** search keys are stored in certain order
  - **hash index (散列索引):** search keys are distributed uniformly across “buckets” using a “hash function”

索引评价标准

- **数据查询类型**
  - 支持的数据访问类型，如找到具有特定属性值的所有记录(Equal Query，等值查询)、找到属性值在某个特定范围内的所有记录(Range Query，范围查询)
- **数据访问时间**
  - 在查询中使用索引找到特定数据所需时间
- **数据插入时间**
  - 插入新数据项的时间，包括: 找到插入位置的时间 + 更新索引结构的时间
- **数据删除时间**
  - 删除一个数据项的时间，包括: 找到待删除项的时间 + 更新索引结构的时间
- **存储空间开销**
  - 索引结构占用的额外存储空间

目录

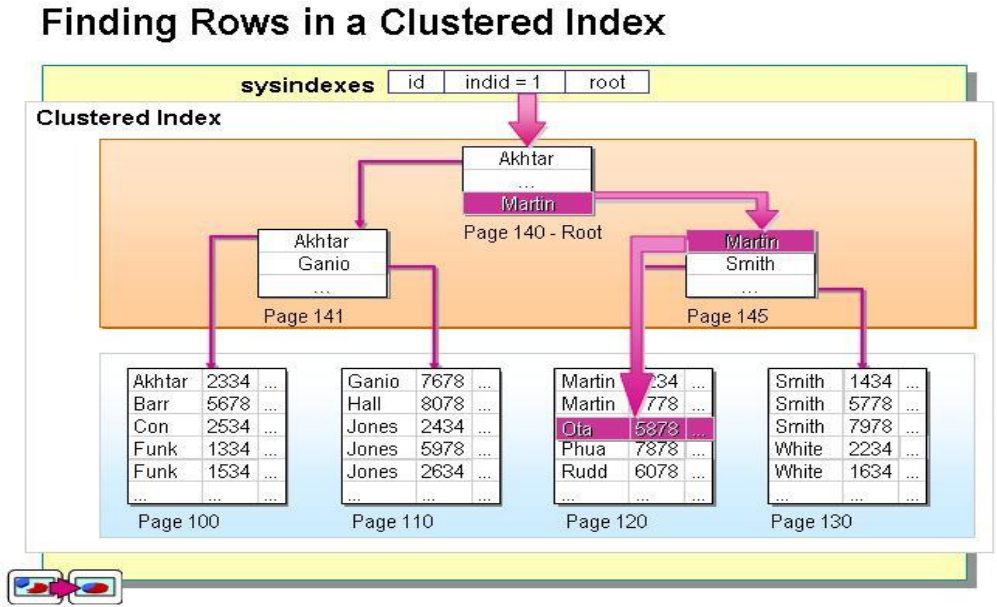
- 基本概念
- 顺序索引
- B^+树和B树索引
- 散列索引
- 多码访问
- 索引创建

顺序索引（Ordered Indexing）

- **Ordered index**
  - Index entries are sorted on the search key value
  - Including primary index and secondary index
- **Primary index (主索引) / clustering index 聚集索引**
  - 包含记录的数据文件按某个搜索码指定的顺序排序，该搜索码对应的索引也称为clustering index
- **Secondary index (辅助索引) / no-clustering index (非聚集索引)**
  - Search key specifies an order different from the sequential order of the data file
- **Index-sequential file (索引顺序文件)**
  - Ordered sequential file with a primary index

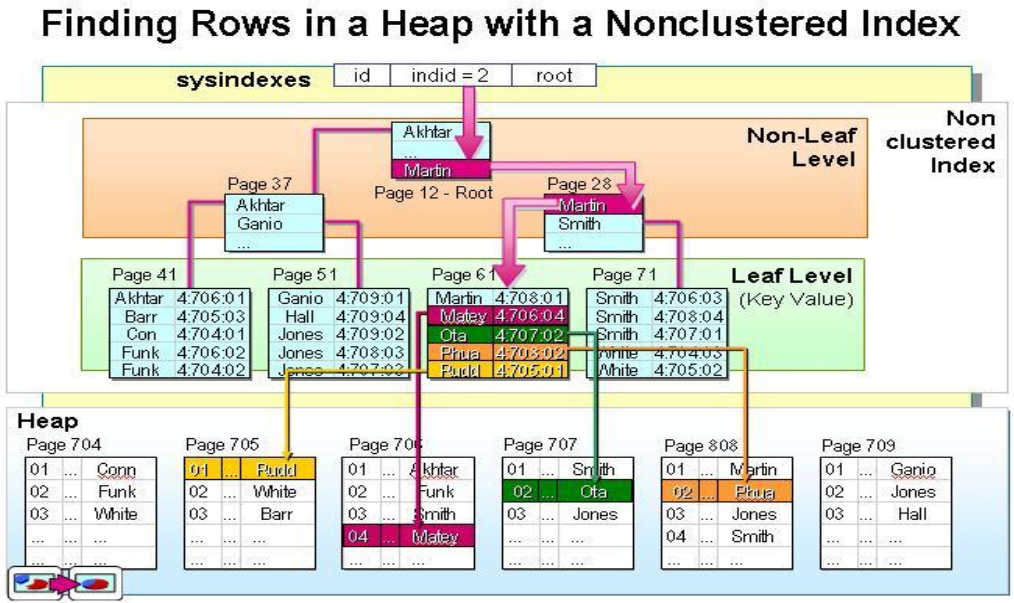
聚集索引

- 树形结构聚集索引的叶节点可以是数据节点，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（思考：为什么？）



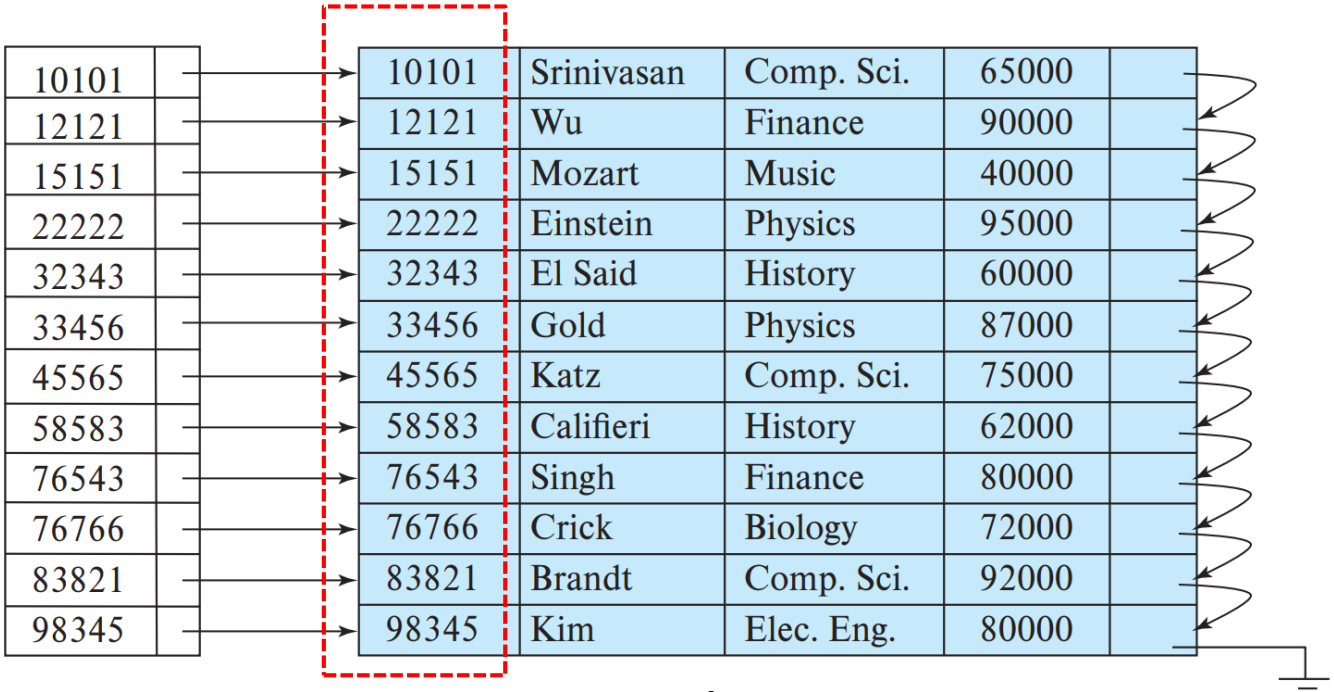
非聚集索引

- 树形结构非聚集索引的叶节点仍然是索引节点，通过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



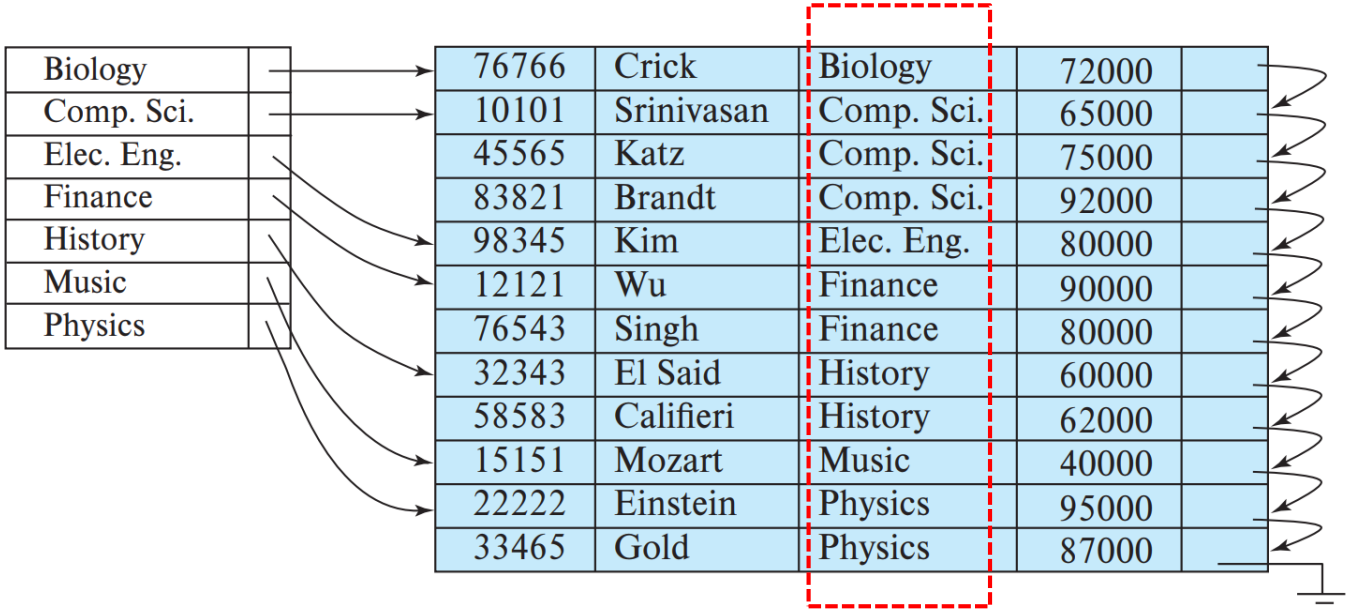
稠密索引

- **Dense index (稠密索引)**
  - Index record appears for every search-key value in the file



稠密索引（续）

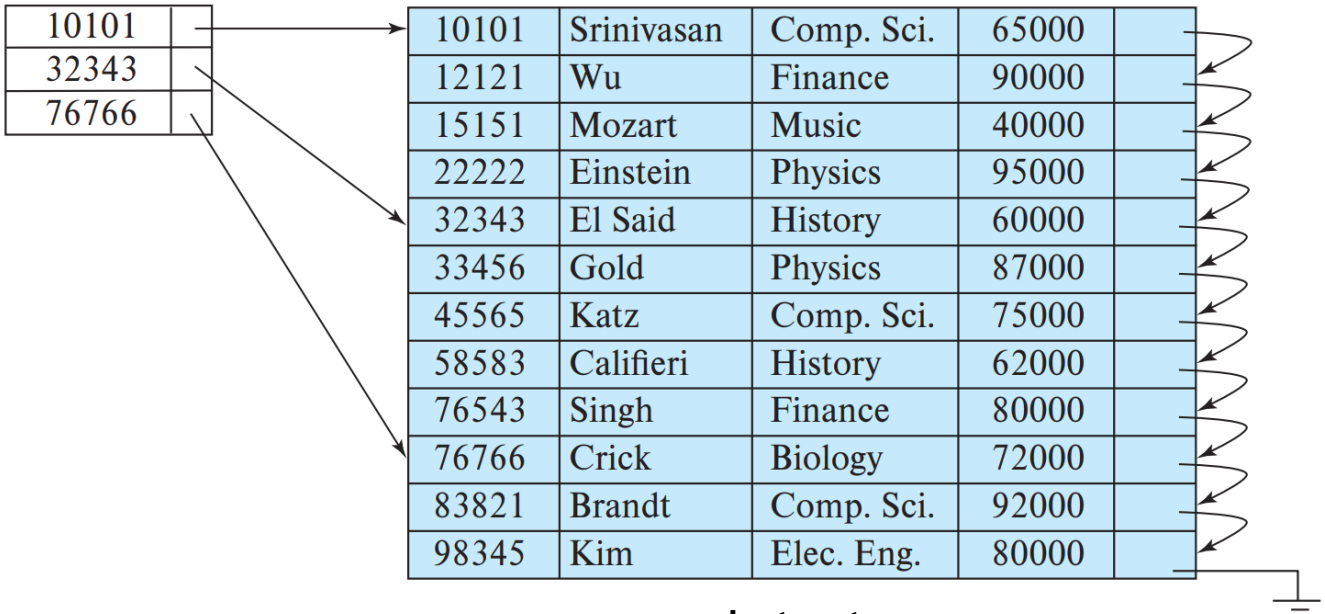
- **Dense index (稠密索引)**
  - Index record appears for every search-key value in the file



Instructor

稀疏索引

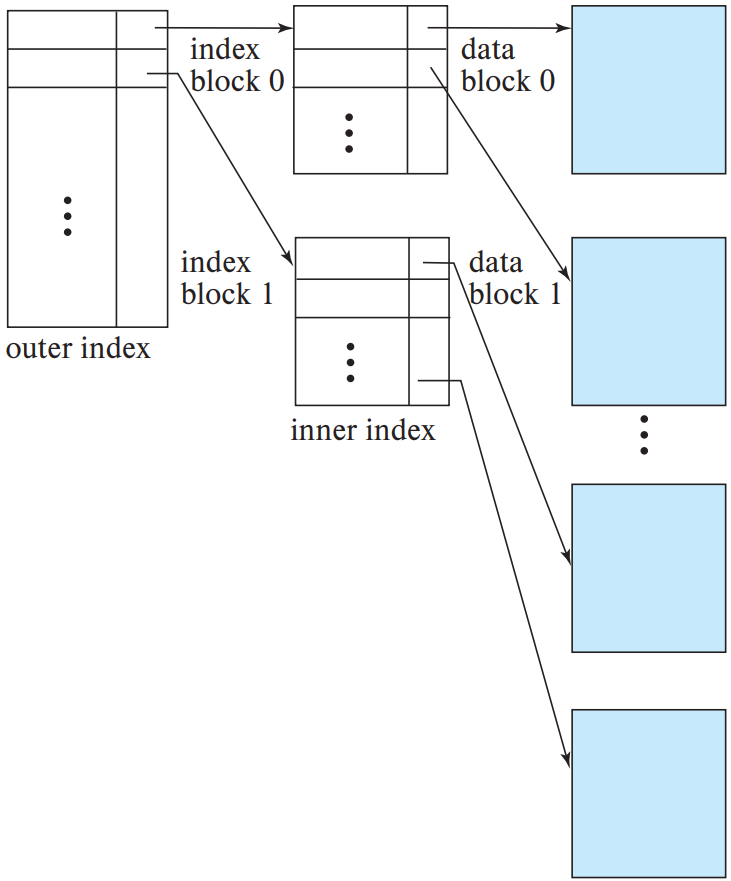
- **Sparse Index (稀疏索引)**
  - Contain index records for only some search-key values when data records are sequentially ordered on search-key (**思考：为什么?**)



多级索引（Multilevel Index）

- If primary index does not fit in memory, data access becomes expensive
- To reduce the number of disk accesses to index records, treat primary index as a sequential file and construct a sparse index on it
  - **inner index** – the primary index file, could be dense or sparse
  - **outer index** – a sparse index of primary index (**思考：为什么是稀疏索引？**)
- If outer index is still too large to fit in main memory, yet another level of index can be created, and so on

多级索引（续）

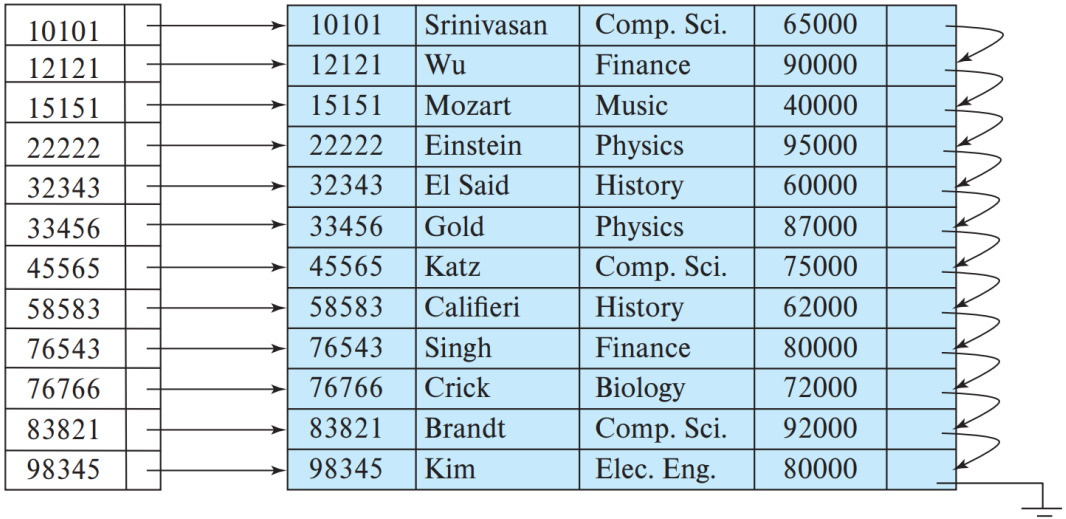

index
block 0
data
block 0
index
block 1
data
block 1
outer index
inner index

稠密索引 vs. 稀疏索引

- **To locate a record with search-key value K**
  - **Dense index**
    - Find the index record with search-key value = K
  - **Sparse index**
    - Find index record with the 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**

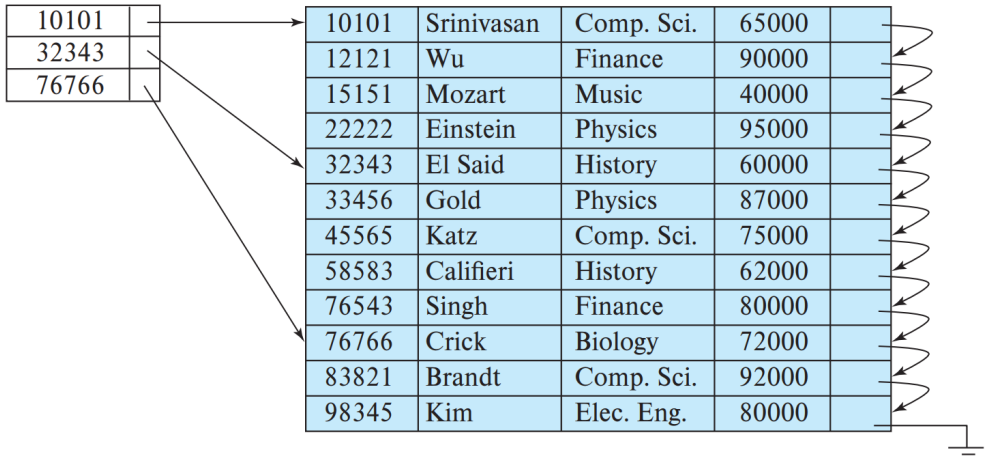
索引维护：删除

- **Single-level index deletion**
  - **Dense indices** – deletion of search-key in index is similar to file record deletion



索引维护：删除（续）

- **Single-level index deletion**
  - **Sparse indices**
    - if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the **next search-key value** in the file
    - if the next search-key value already has an index entry, the entry is deleted instead of being replaced



索引维护：插入

- **Single-level index insertion**
  - Perform a lookup using the search-key value
  - **Dense indices** – if the search-key value does not appear in the index, insert it
  - **Sparse indices** – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created. In this case, the **first search-key value** appearing in the new block is inserted into the index
- **Multilevel insertion/deletion**
  - Extensions of the single-level algorithms

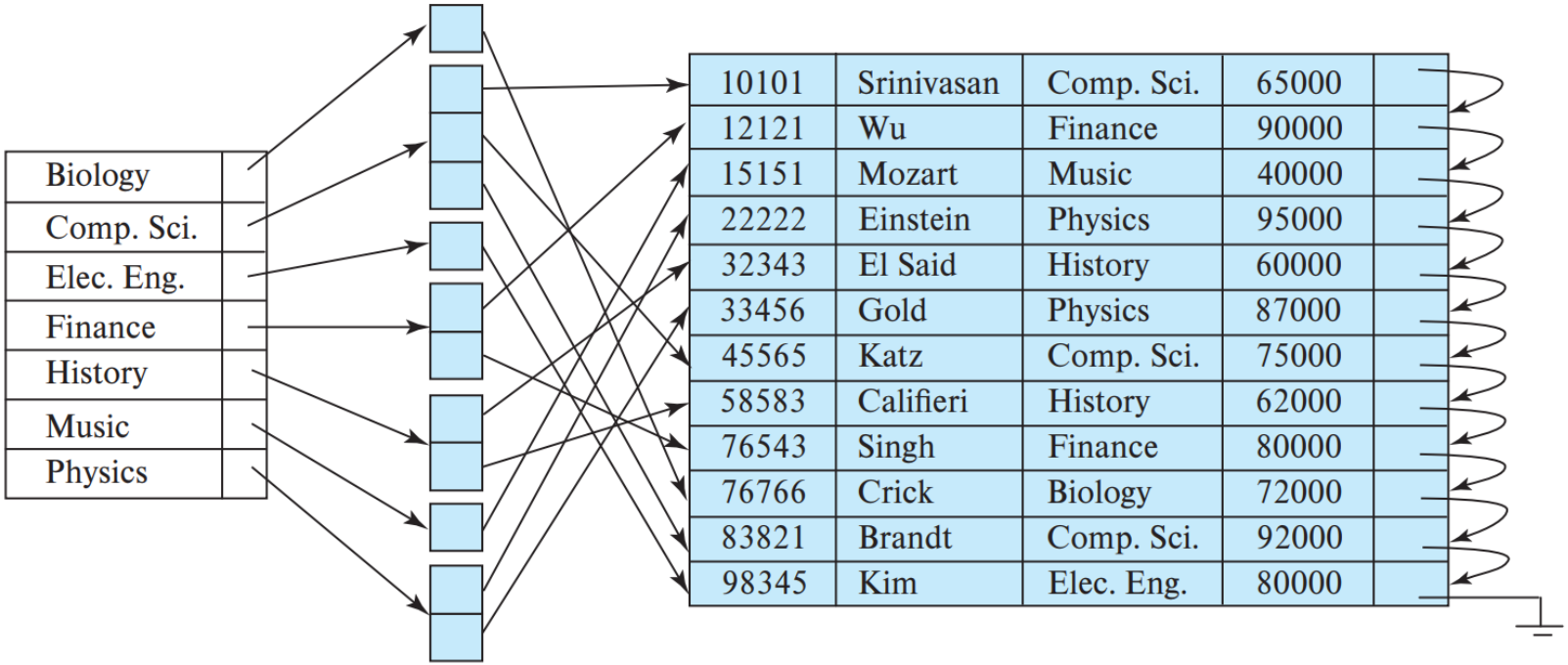
稠密索引 vs. 稀疏索引

- **Space and maintenance for insertions and deletions**
  - Sparse index needs less space and less maintenance overhead for insertions and deletions
  - **One good choice**: sparse index with an index entry for every block in file, corresponding to the least search-key value in the block

非聚集索引

- **To find all the records with values in a certain range**
  - **Example 1**: Considering the student relation stored sequentially by student id, we may want to find all the students in a particular department
  - **Example 2**: To find all the students with a specified GPA or a range of GPA
- **Secondary index**
  - Build a secondary index with an index record for **each search-key value**
  - Index record points to a **bucket** that contains **pointers** to all the actual records with that particular search-key value

Account表中Balance属性上的非聚集索引



bucket

聚集索引和非聚集索引

- Secondary indices have to be dense (**思考：为什么?**)
- When a data file is modified, the index on the file must be updated. Updating indices imposes overhead on database modification
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive (**思考：为什么?**)
  - each record access may fetch a new block from disk

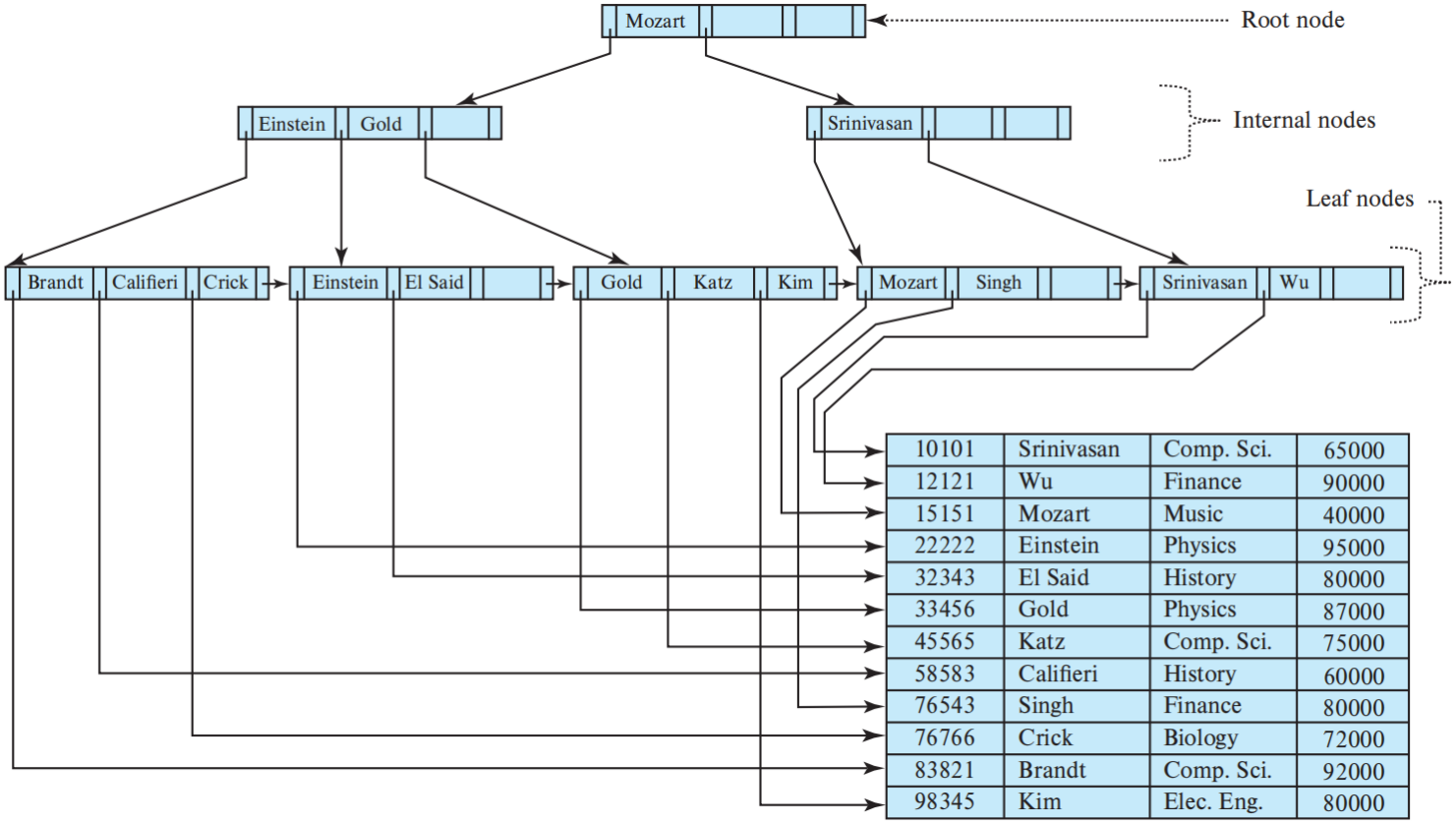
目录

B^+-树索引

B^+-tree is an alternative to indexed-sequential file

- Disadvantage of indexed-sequential file
  - Performance degrades as file grows, since many **overflow blocks** (溢出块) get created. Periodic reorganization of entire file is required
- **B^+-tree index file**
  - **Advantage**: automatically reorganizes itself with small and local changes, in the face of insertions and deletions. Reorganization of entire file is not required to maintain performance
  - **Disadvantage**: extra insertion and deletion overhead, and more space overhead
  - B^+-tree is used widely since its advantages outweight the disadvantages

B+-树示例



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⁺-树索引（续）

- **Typical B⁺-tree node**

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

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

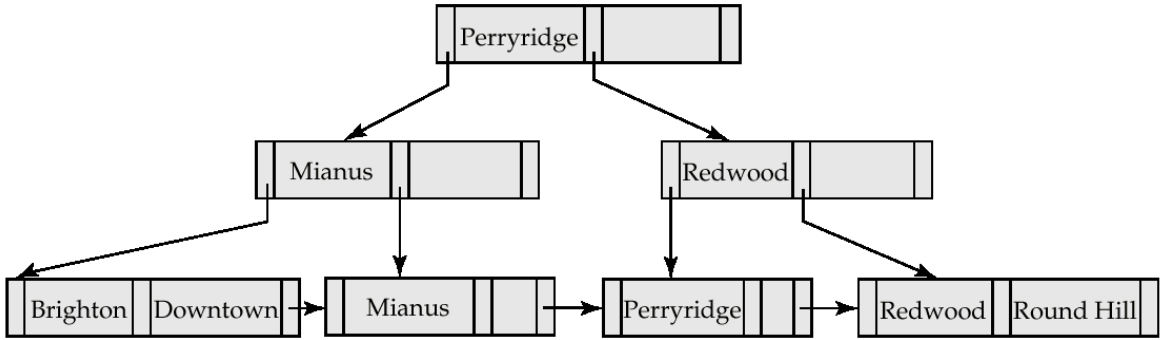
  - P_i: the pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes)

B⁺-树索引（续）

- B⁺-tree is a rooted tree (有根树) with the following properties:
  - B⁺-tree is a **balanced tree (平衡树)** 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 $n-1$ search-key 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** search-key values

B⁺-树示例

- 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 should have at least 2 children



B⁺-tree for *account* file (*n* = 3)

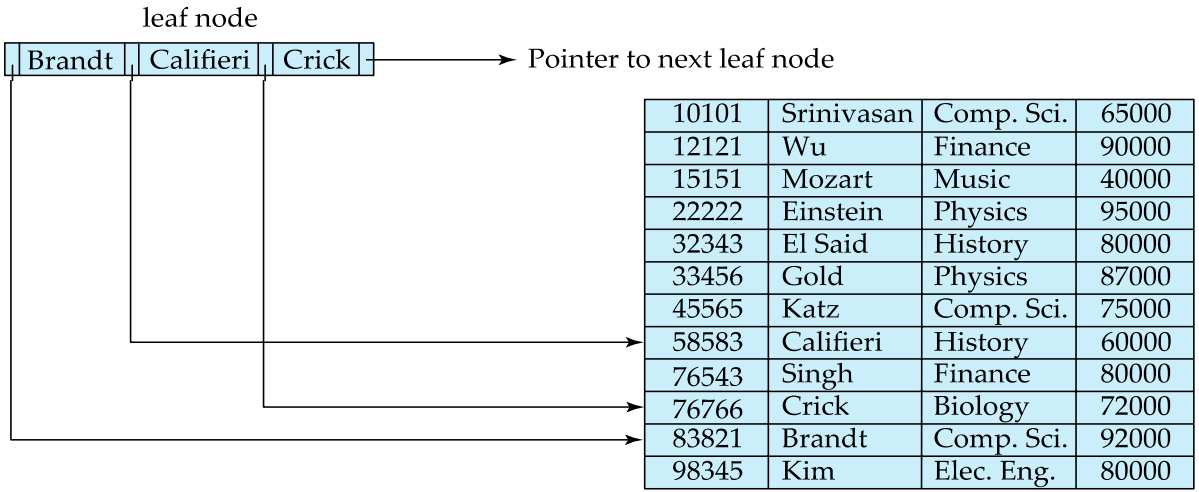
B⁺-树的叶子节点

- **Properties of a leaf node**

P_1	K_1	P_2	$\cdots$	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 with search-key value K_i. Bucket structure is needed if the search-key does not form a primary key (Why?)
  - P_n points to the next leaf node in search-key order

n=4



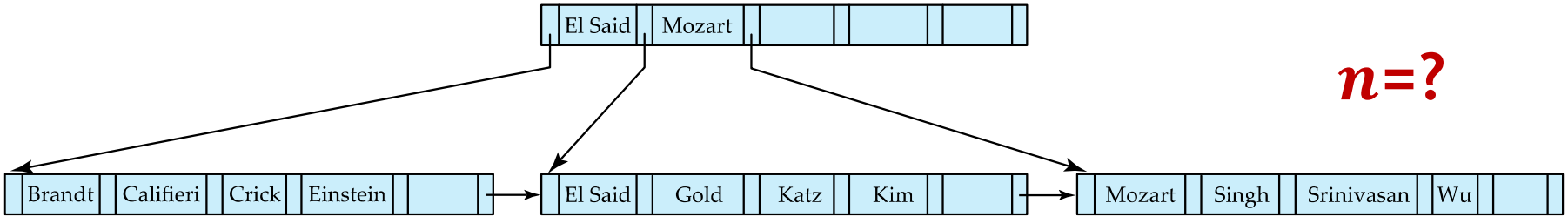
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000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

instructor file

B+-树的非叶子节点

- 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⁺-树示例



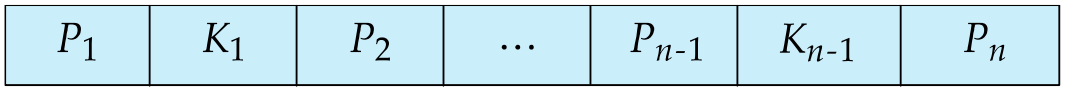
- B⁺-tree for ***instructor*** file (n = 6)
  - Leaf nodes must have between 3 and 5 values ($\lceil (n-1)/2 \rceil$ and $n-1$)
  - Non-leaf nodes other than root must have between 3 and 6 children ($\lceil n/2 \rceil$ and n)
  - Root must have at least 2 children

B⁺-树的特点

- 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 small number of levels, and search can be conducted efficiently
  - If there are K search-key values in the file, the tree height is about $\lceil log_{n/2}(K) \rceil$
    - 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

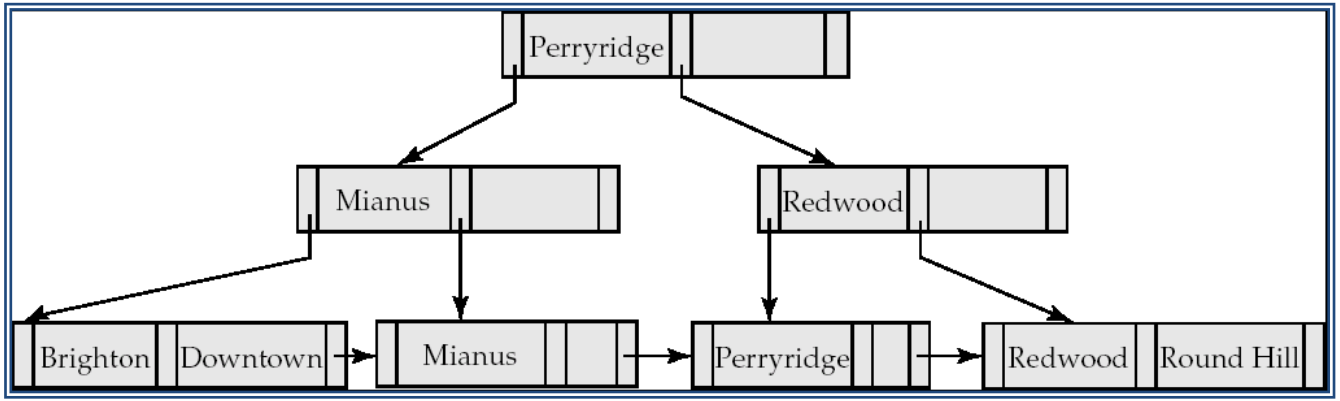
B+-树查询

- **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. For some i, if key $K_i = k$, follow pointer P_i to the desired record or bucket. Otherwise, no record with search-key value k exists



B+-树查询（续）

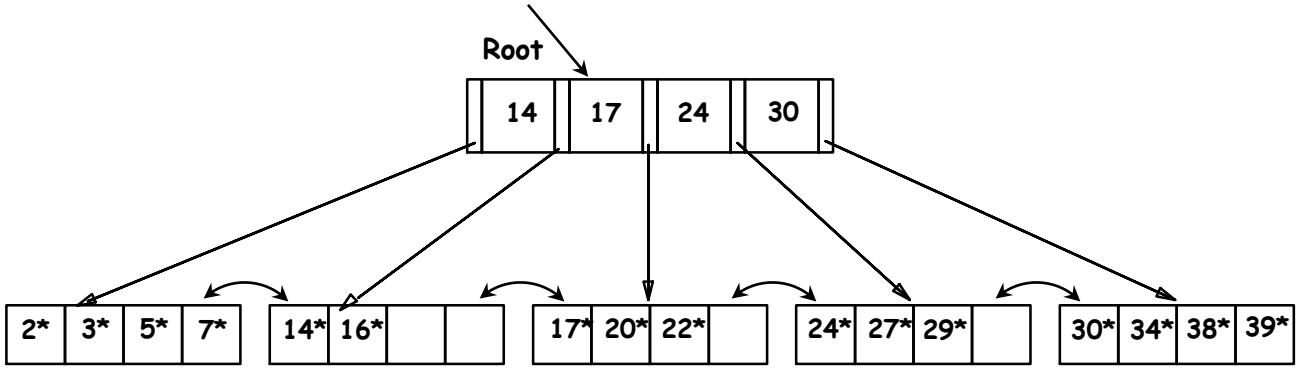
- Search begins at root, and key comparisons direct it to a leaf node
  - Search for Perryridge



P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

B+-树查询（续）

- Search begins at root, and key comparisons direct it to a leaf node
  - Search for ***5*, 15*, all data entries >= 24****



P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

B⁺-树查询（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 has the same size as a disk block, typically 4 KB, and n is typically around 100 (40B per index entry)
    - For a B⁺-tree with 1 million search key values and n = 100, at most $\log_{50}(1{,}000{,}000)$ = 4 nodes are accessed in a query
    - 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 query
    - The above difference is significant since every node access may need a disk I/O, costing around 10 ms

B^+-树的插入

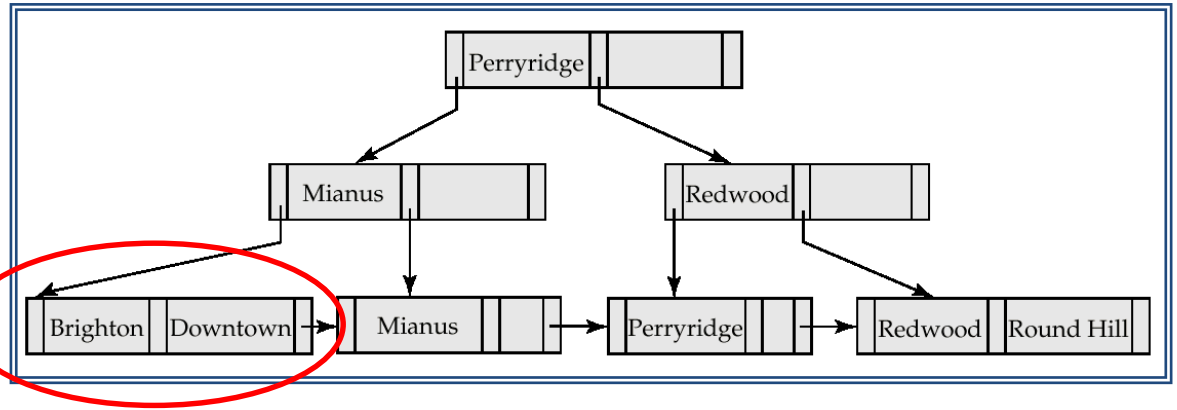
- **Find the leaf node in which the search-key value would appear**
  - If the search-key value is already in the leaf node
    - Add the record to the data file, and insert its pointer into the corresponding pointer bucket
  - If the search-key value is not in certain node, add the record to the data file and create a new bucket. Then:
    - If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
    - Otherwise, split the node along with the new (key-value, pointer) entry

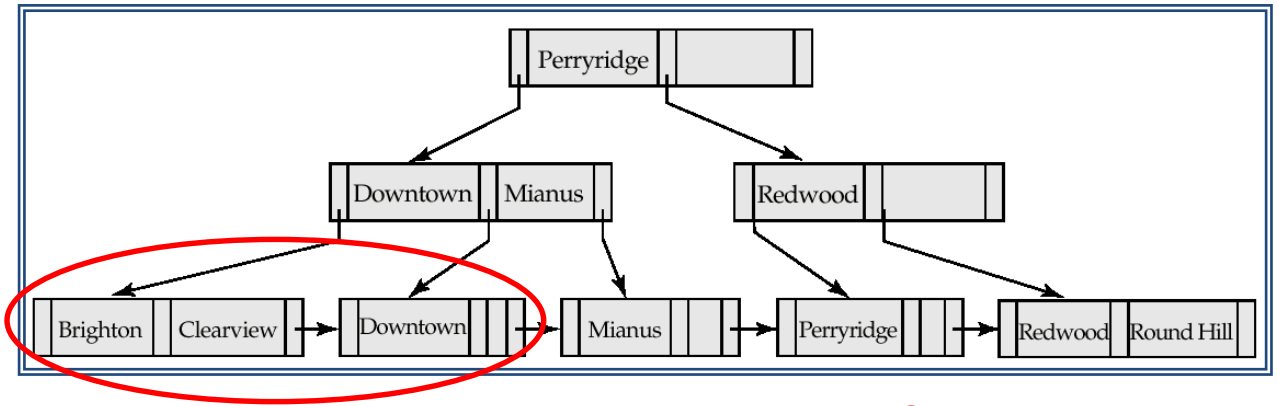
B⁺-树的插入（续）

- **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 p, and let k be the least key value in 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 will be split, thus increasing the height of the tree by 1

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

B+-树的插入（续）





B+-Tree before and after the insertion of "Clearview"

B⁺-树的插入（续）

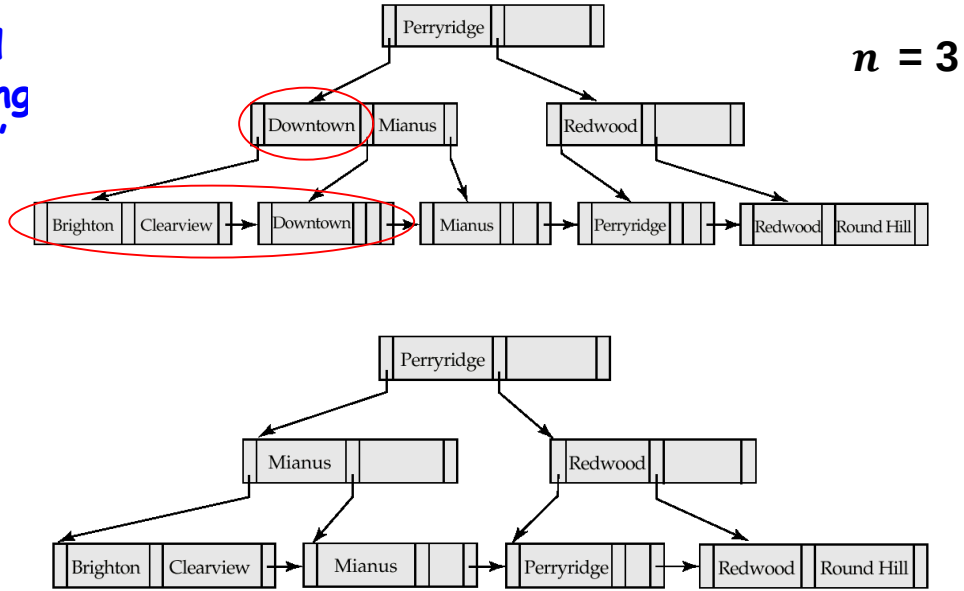
- **Splitting a non-leaf node:** when inserting (k, p) into an full internal node N
  - Copy N to an in-memory area M with space for $n + 1$ pointers and n keys
  - Insert (k, p) into M
  - Copy $P_1, K_1, \ldots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ from M back into node N
  - Copy $P_{\lceil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

B⁺-树的删除（续）

- Find the record to be deleted, and remove it from the data file and from the pointer 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) from its parent, recursively using the above procedure, where P_i is the pointer to the deleted node

B+-树的删除示例

Before and after deleting "Downtown"



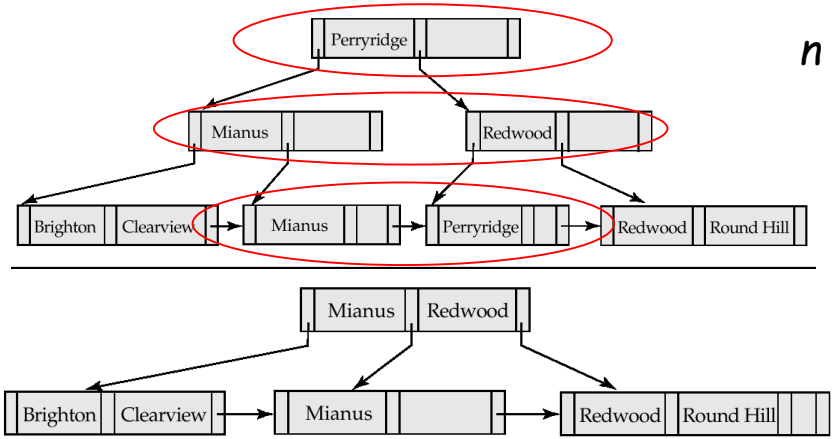
- Deleting "Downtown" causes the merging of under-full leaves
- The removal of the leaf node containing "Downtown" did not result in its parent having too little pointers. So the cascaded deletions stopped with the deleted leaf node's parent

B^+-树的删除

- 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+-树的删除示例

Deletion of "Perryridge"

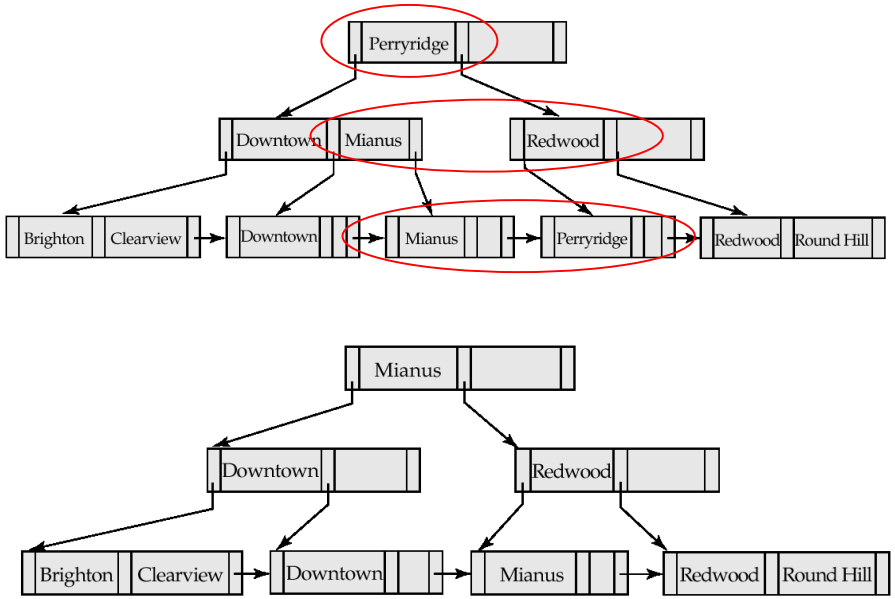


$n = 3$

- Node with "Perryridge" becomes underfull (actually empty, in this special case) and merged with its sibling
- As a result "Perryridge" node's parent became underfull, and was merged with its sibling (and an entry was deleted from their parent)
- Root node then had only one child, and was deleted and its child became the new root node

B+-树的删除示例

Deletion of "Perryridge"



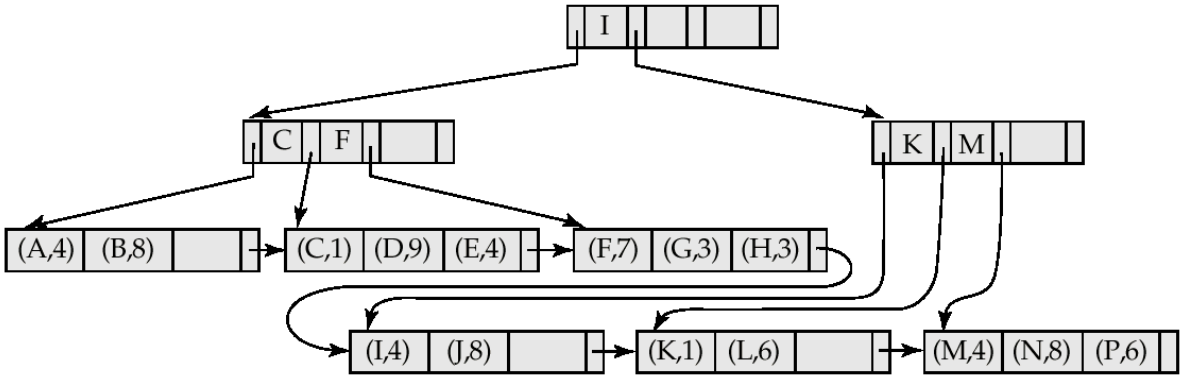
- Parent of leaf containing Perryridge became underfull, and borrows a pointer from its left sibling
- Search-key value in the parent's parent changes as a result

B^+-树文件组织

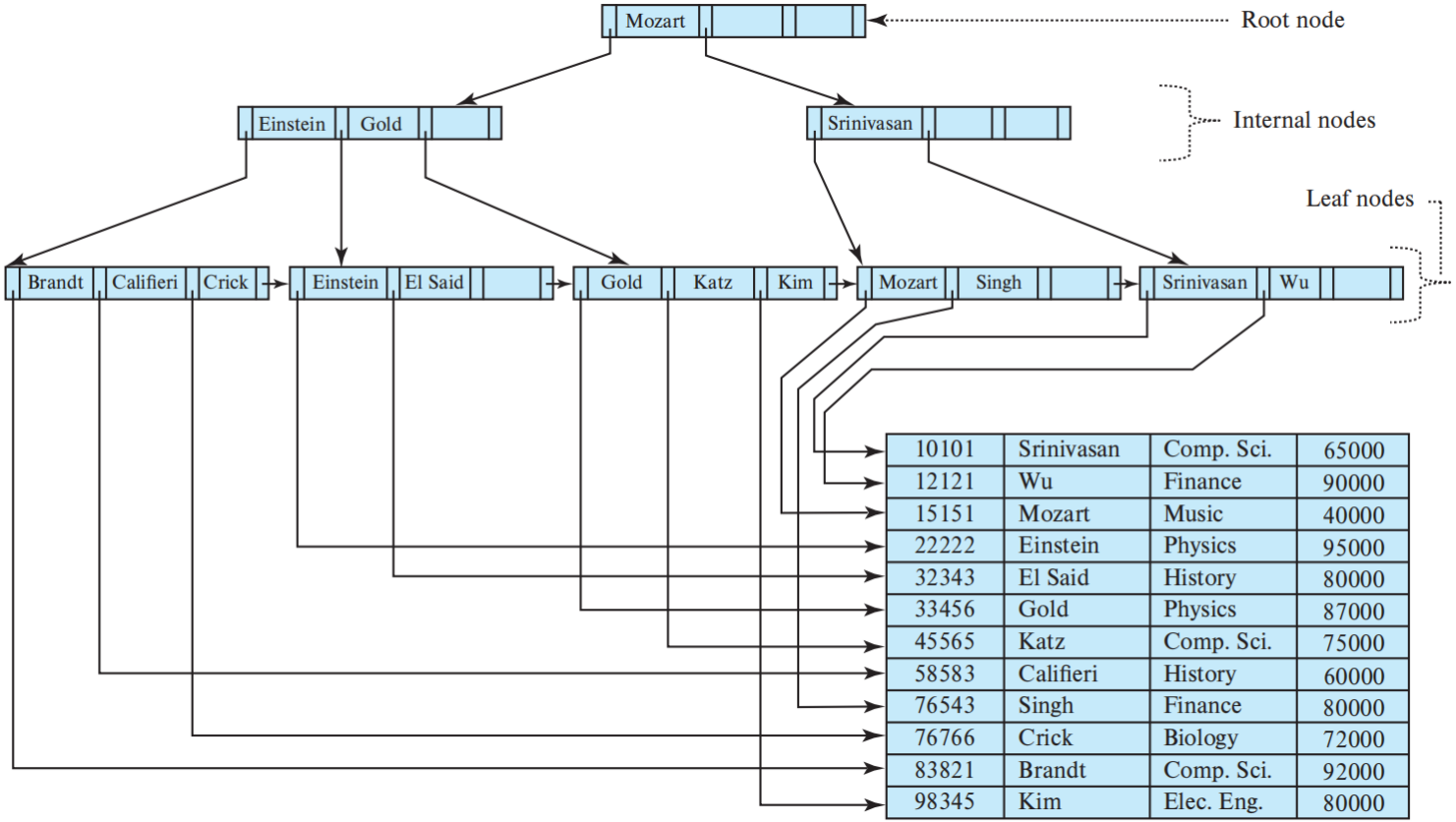
- Index file degradation (退化) problem is solved by using B^+-Tree indices. Data file degradation problem is solved by using B^+-Tree File Organization (B^+树文件组织)
- The leaf nodes in a B^+-tree file organization store records, instead of pointers
- Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a non-leaf node
- Leaf nodes are still required to be half full (思考：是否一定可行？)
- Insertion and deletion are handled in the same way as the insertion and deletion of entries in a B^+-tree index

B^+-树文件组织（续）

- Good space utilization is important since records use more space than pointers.
- To improve space utilization, involve more sibling nodes in redistribution
  - Involving 2 siblings or more in redistribution to avoid split / merge where possible



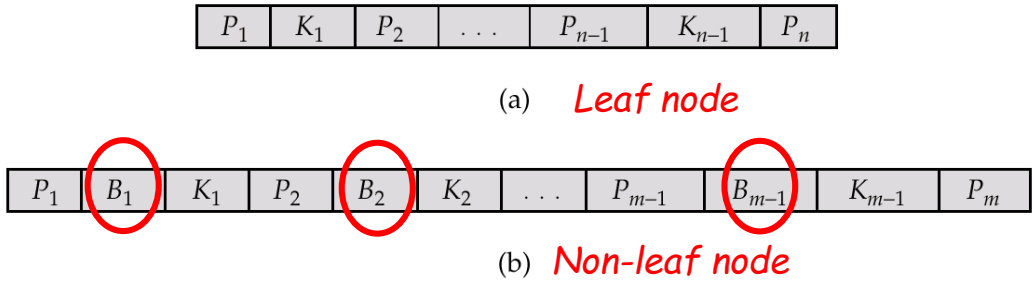
B+-树示例



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

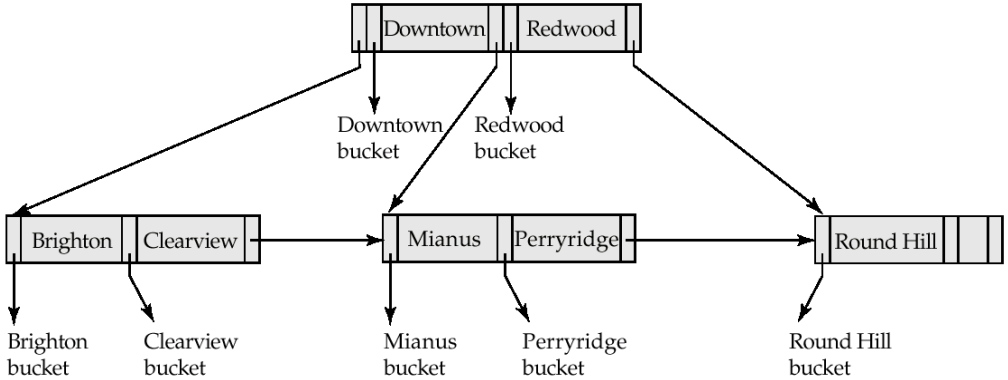
B-树索引

- 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, and an additional pointer field for each search key in a non-leaf node is included
- General B-tree leaf node and non-leaf node

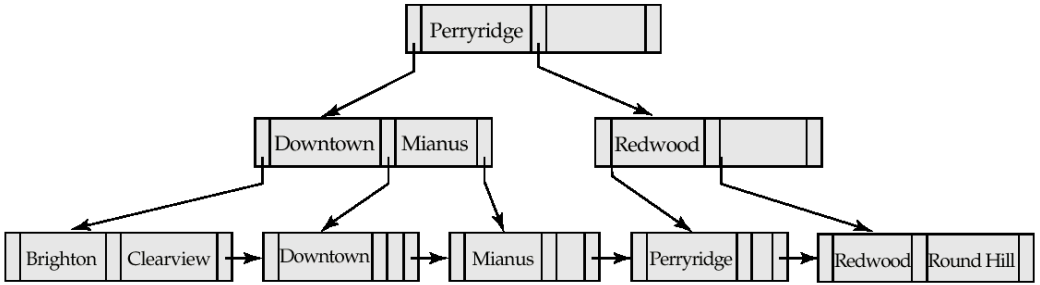


- Non-leaf node – pointers B_i are the bucket or file record pointers

B-树索引（续）



B-tree (above) and B+-tree (below) on the same data



B-树索引（续）

- **Advantages of B-tree indices**
  - Use less tree nodes than B^+-Tree
  - Possible to find the search-key value before reaching leaf nodes

- **Disadvantages of B-tree indices**
  - Only a small fraction of all search-key values are found early
  - Non-leaf nodes are larger, so fan-out is reduced. Thus B-trees typically have greater depth than B^+-tree
  - Insertion and deletion are more complicated than in B^+-trees
  - Implementation is harder than B^+-trees

- Typically, the advantages of B-trees do not outweigh disadvantages

目录

- 基本概念
- 顺序索引
- B^+树和B树索引
- 散列索引
- 多码访问
- 索引创建

静态散列（Static Hashing）

- **Bucket**
  - a unit of storage containing one or more records (a bucket is typically a disk block)
  - the corresponding bucket of a record is directly obtained from its search-key value using a hash function
- **Hash function h**
  - a function from the set of all search-key values K to the set of bucket addresses
  - used to locate records for access, insertion as well as deletion
- **Note:**
  - records with different search-key values may be mapped to the same bucket
  - the entire bucket has to be searched sequentially to locate a record

散列文件组织

- Hash file organization of *account* file, using *branch-name* as key (See figure in next slide)
  - There are 10 buckets
    - The binary representation of the i-th character is assumed to be the integer i
    - The hash function returns the sum of the binary representations of the characters modulo 10
    - E.g.
      h(Perryridge) = 125 mod 10 = 5
      h(Round Hill) = 113 mod 10 = 3
      h(Brighton) = 93 mod 10 = 3

散列文件组织示例

Hash file organization of *account* file, using *branch-name* as key

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

散列函数

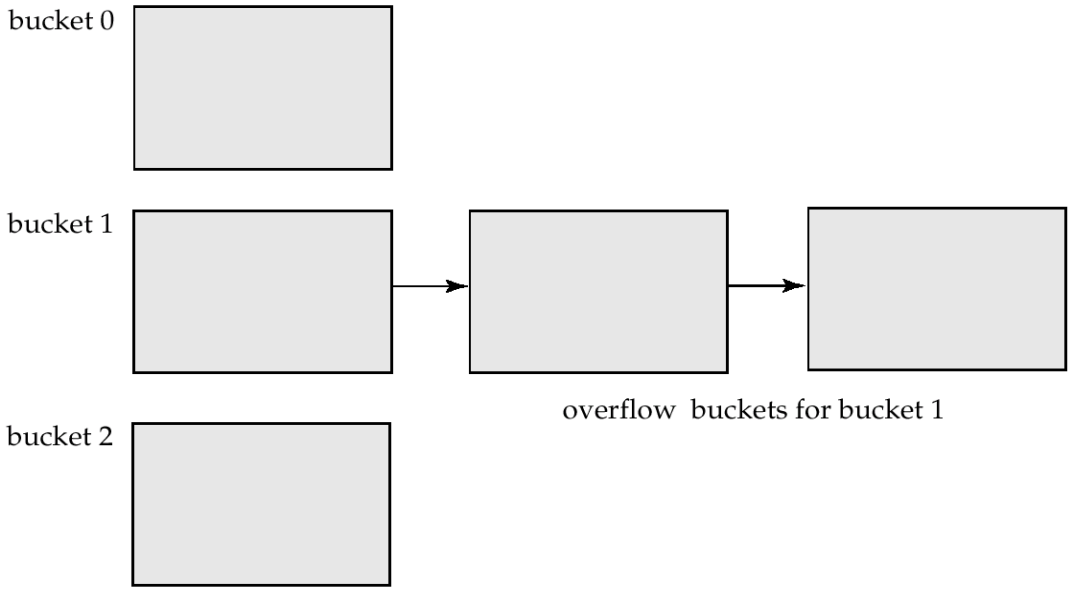
- **Good hash function**
  - **Uniform**: each bucket is assigned the same number of search-key values from the set of all possible values
  - **Random**: each bucket has the same number of records assigned to it irrespective of the actual distribution of search-key values in the file
- **Worst hash function**
  - maps all search-key values to the same bucket
- Typical hash functions perform computation on the internal binary representation of the search-key

溢出桶的处理

- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records due to two reasons:
    - multiple records have the same search-key value
    - hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated. It is handled by using overflow buckets

溢出桶的处理（续）

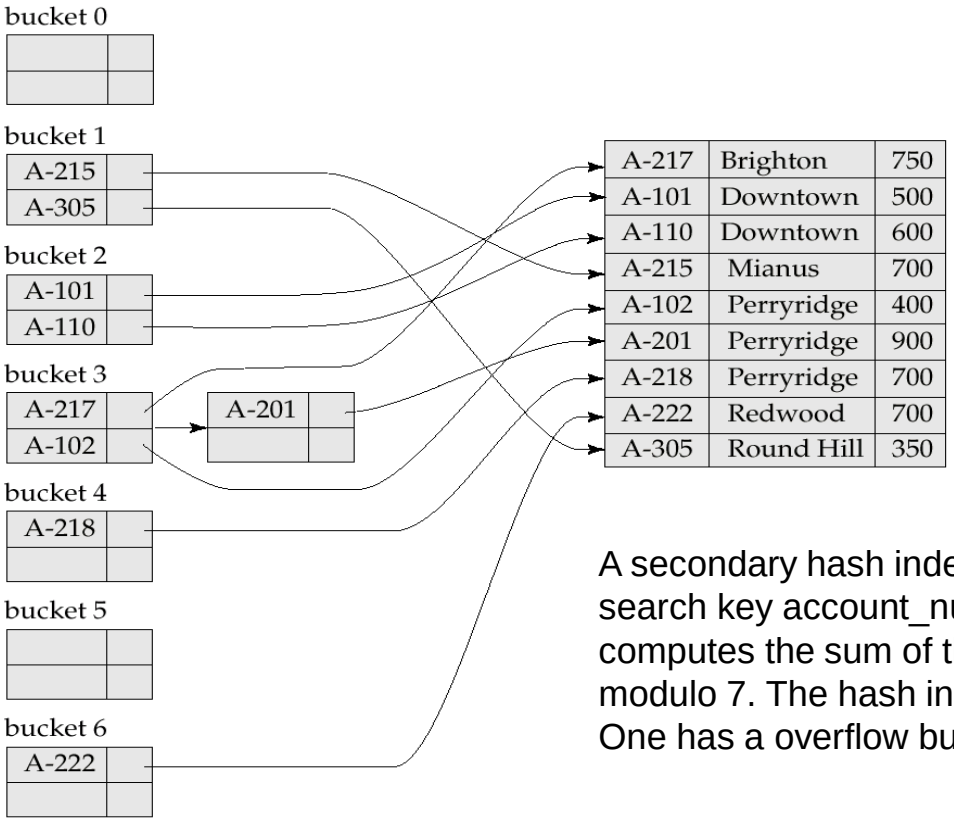
- Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list



散列索引

- Hashing can be used not only for file organization, but also for index-structure creation
- A hash index organizes the search keys, with their associated record pointers, into a hash file structure
- Strictly speaking, hash indices are always **secondary indices**
  - If the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary
  - Hash index refers to both secondary index structures and the hash organized files

散列索引示例



A secondary hash index on the account file, for the search key account_number. The hash function computes the sum of the digits of the account number modulo 7. The hash index has 7 buckets, each of size 2. One has a overflow bucket.

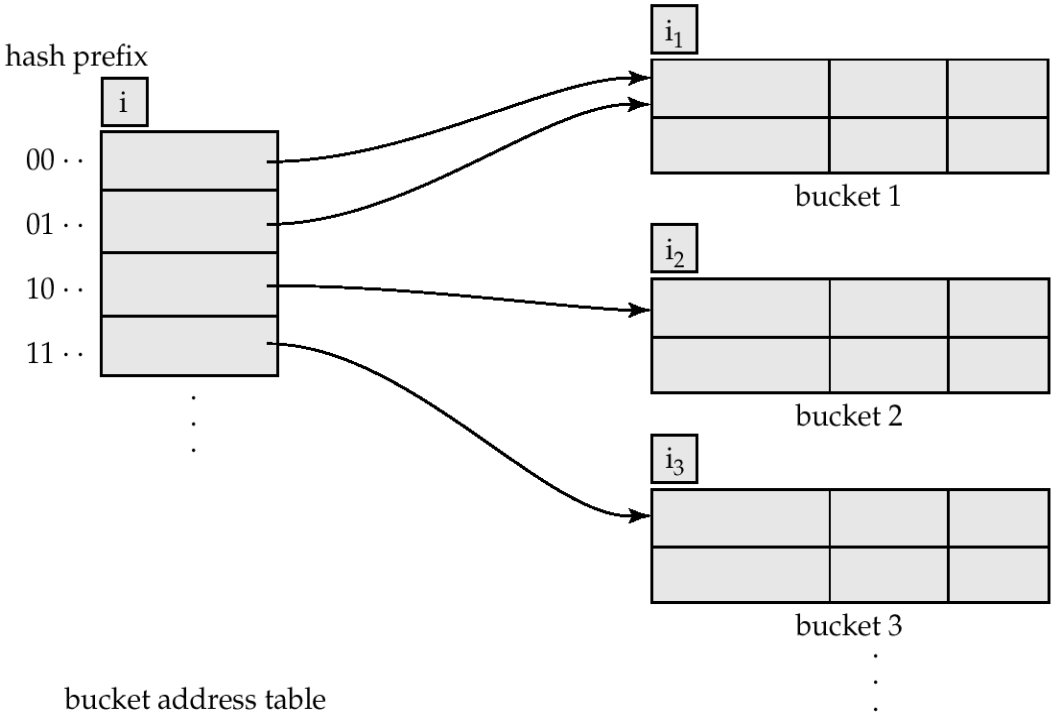
静态散列的不足

- 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

动态散列（Dynamic Hashing）

- **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, the actual number of buckets is < 2^i
    - The number of buckets also changes dynamically due to coalescing and splitting of buckets.

可扩展散列结构


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

可扩展散列结构（续）

- 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:
  - Compute $h(K_j) = X$
  - 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)

可扩展散列结构的更新

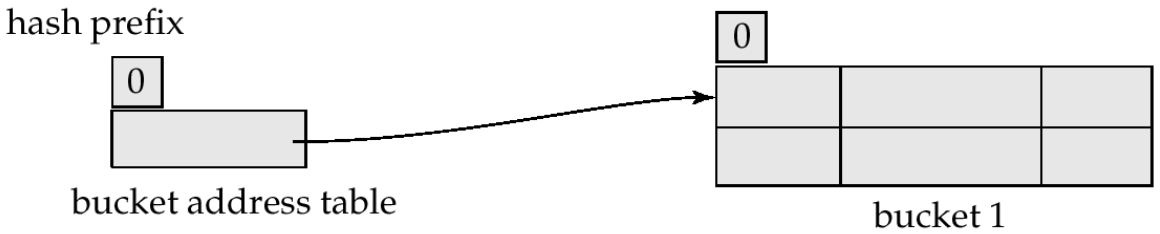
- **To split a bucket j when inserting record with search-key value K_j:**
  - If $i > i_j$ (more than one pointer to bucket j)
    - allocate a new bucket z, and set i_j and i_z to the old $i_j + 1$
    - make the second half of the bucket address table entries pointing to j to point to z
    - remove 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

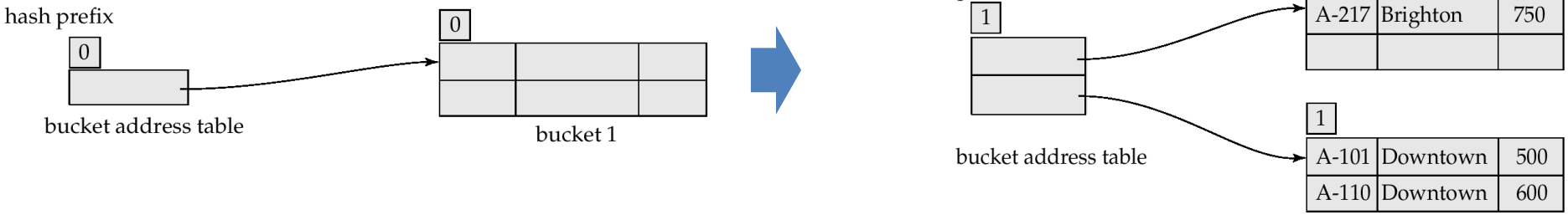
branch-name	h(*branch-name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111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



Initial Hash structure, bucket size = 2

示例（续）

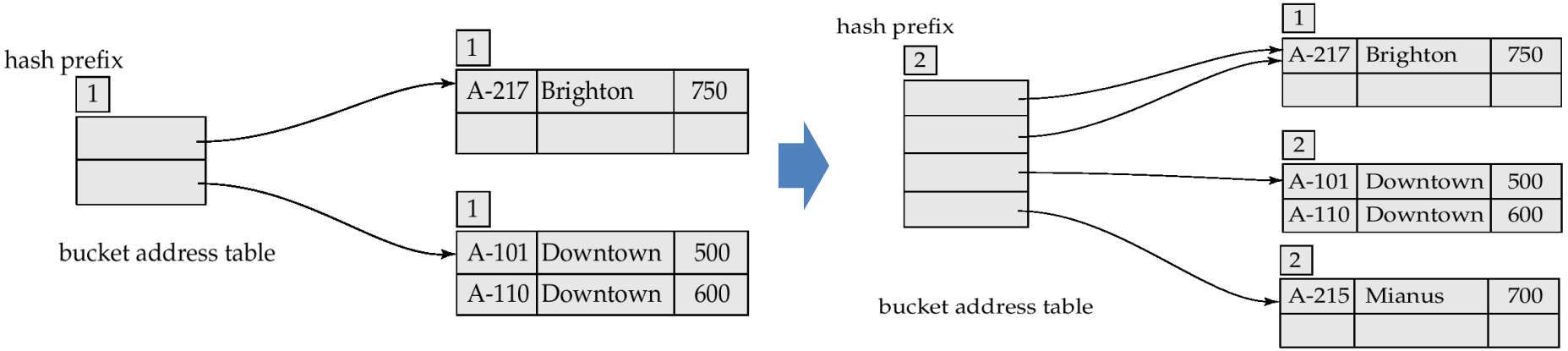
- Hash structure after insertion of one Brighton and two Downtown records



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

示例（续）

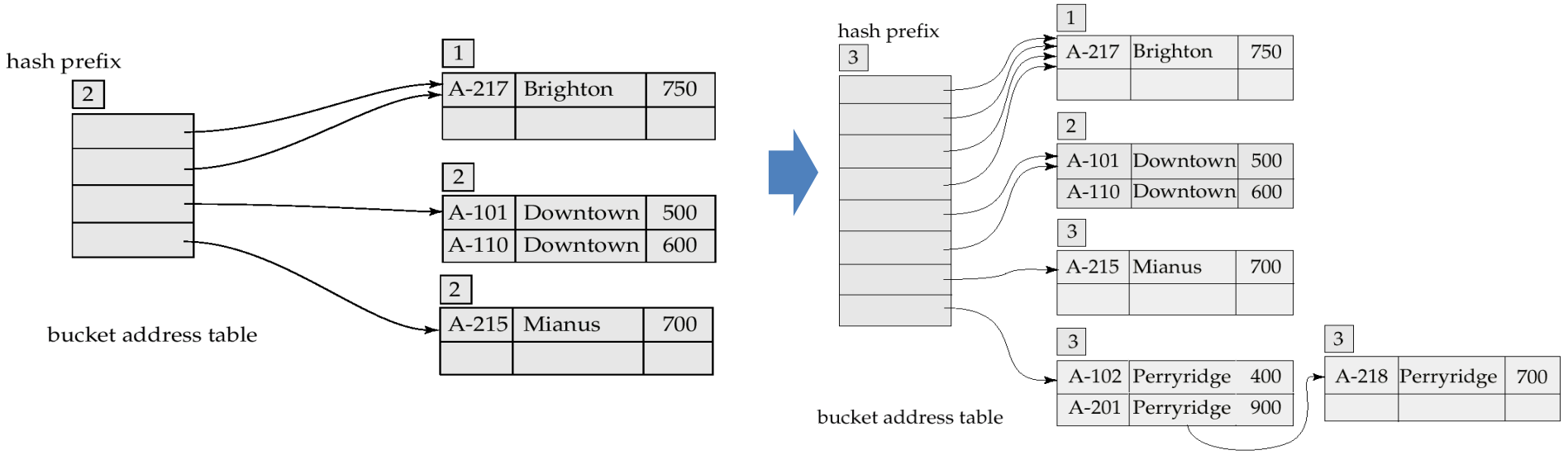
- **Hash structure after insertion of Mianus record**



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

示例（续）

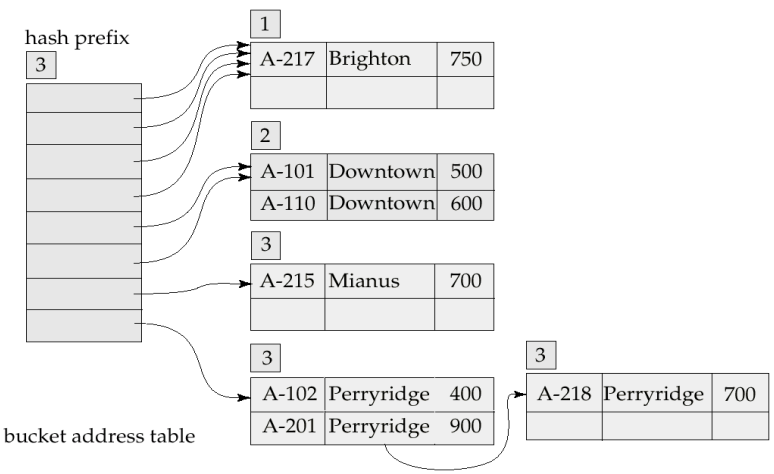
Hash structure after insertion of three Perryridge records

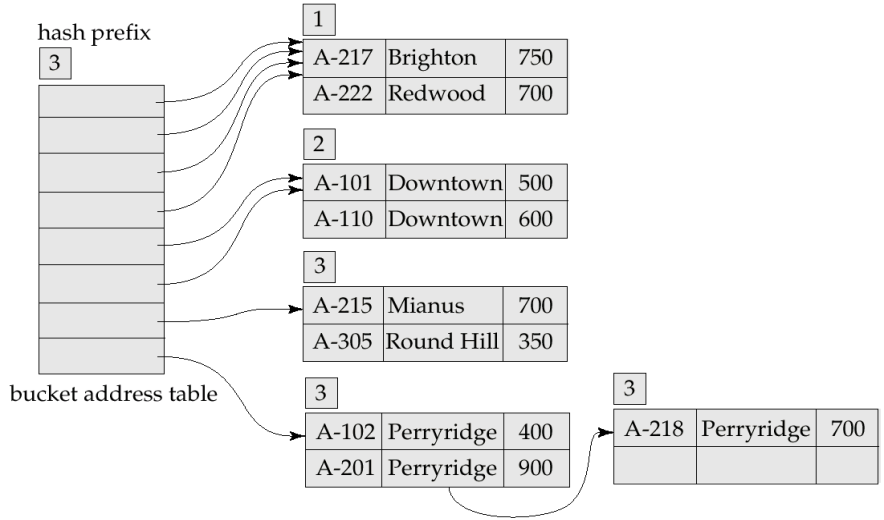


branch-name	h(*branch-name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111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

示例（续）

- **Hash structure after insertion of Redwood and Round Hill records**





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

可扩展散列 vs. 其他模式

- Benefits of extendable hashing:
  - Hash performance does not degrade with growth of file
  - Minimal space overhead
- Disadvantages of extendable hashing
  - Extra level of indirection to find desired record
  - Bucket address table may itself become very big (larger than memory)
    - Need a tree structure to locate desired record in the structure!
  - Changing size of bucket address table is an expensive operation
- Linear hashing is an alternative mechanism which avoids these disadvantages at the possible cost of more bucket overflows

顺序索引 vs. 散列索引

- **Issues for file organizing and indexing**
  - Cost of periodic re-organization
  - Frequency of insertions and deletions
  - Whether optimizing average access time at the expense of worst-case access time
  - **Expected type of queries**
    - Hashing is generally better at retrieving records having a specified value of the key
    - If range queries are common, ordered indices are preferred

目录

多码访问（Multiple-Key Access）

- Use multiple indices for certain types of queries
  - E.g.,

select *account_number*
from *account*
where *branch_name = "Perryridge"* ***and*** *balance = 1000*

- Three possible strategies for processing query using indices on single attributes
  - use index on *branch_name* to find accounts with *branch_name* = "Perryridge", test balances of $1000
  - use index on balance to find accounts with balances of $1000, test *branch_name* = "Perryridge"
  - use *branch_name* index to find pointers to all records pertaining to the Perryridge branch. Similarly use index on balance. Take intersection of both sets of pointers obtained

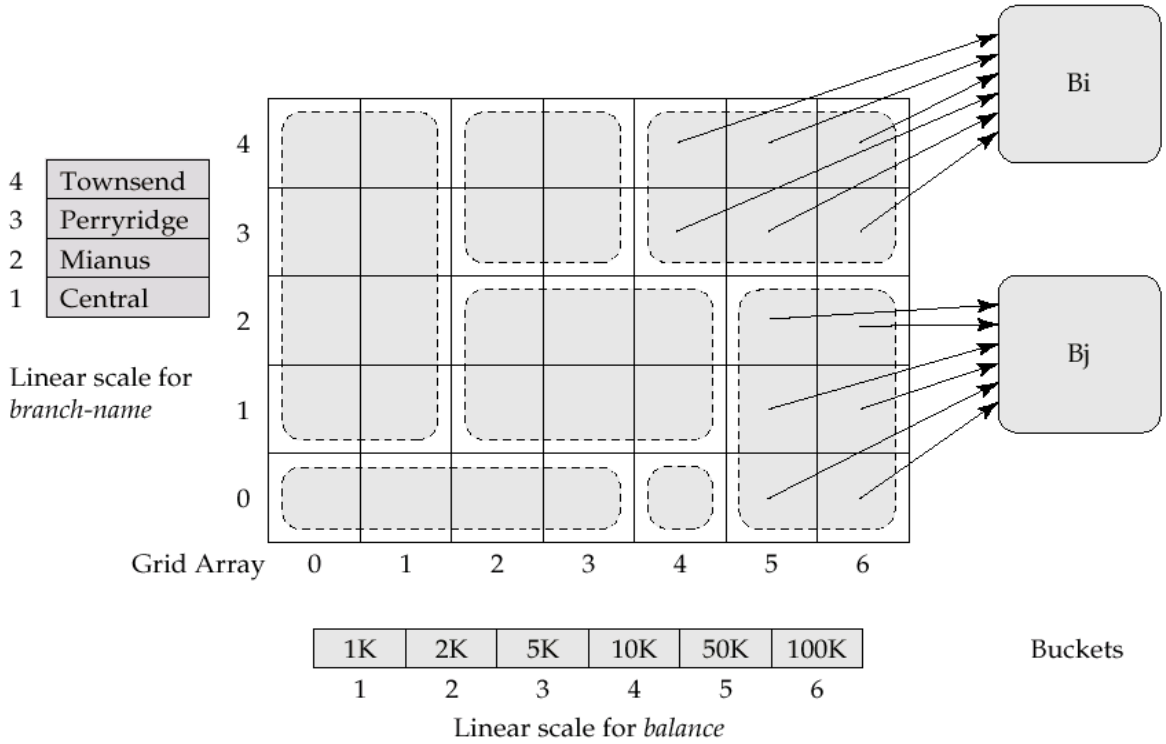
多属性索引

- Suppose we have an index on combined search-key (branch_name, balance)
- With the where clause
  where branch_name = "Perryridge" and balance = 1000
  the index on the combined search-key will fetch only records that satisfy both conditions
- Can also efficiently handle
  where branch_name = "Perryridge" and balance < 1000
- But cannot efficiently handle
  where branch-name < "Perryridge" and balance = 1000
  May fetch many records that satisfy the first but not the second condition, may lead to many I/Os

网格文件（Grid Files）

- Structure used to speed up the processing of multiple search-key queries involving one or more comparison operators
- The grid file has a single grid array and one linear scale for each search-key attribute. The grid array has the number of dimensions equal to number of search-key attributes
- Multiple cells of grid array can point to same bucket
- To find the bucket for a search-key value, locate the row and column of its cell using the linear scales and follow pointer

account表的网格文件



网格文件上的查询

- 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）

- During insertion, if a bucket becomes full, new bucket can be created if more than one cell points to it
  - Idea similar to extendable hashing, but on multiple dimensions
  - If only one cell points to it, either an overflow bucket must be created or the grid size must be increased
- Linear scales must be chosen to uniformly distribute records across cells.
  - Otherwise there will be too many overflow buckets.
- Periodic re-organization to increase grid size will help
  - But reorganization can be very expensive.
- Space overhead of grid array can be high.

位图索引（Bitmap Indices）

- 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

位图索引（续）

- Bitmap indices are useful for queries on multiple attributes
  - not particularly useful for single attribute queries
- Queries are answered using bitmap operations
  - Intersection (and)
  - Union (or)
  - Complementation (not)
- Each operation takes two bitmaps of the same size and applies the operation on the corresponding bits to get the result bitmap
  - E.g., 100110 AND 110011 = 100010
    100110 OR 110011 = 110111
    NOT 100110 = 011001
  - Males with income level L1: 10010 AND 10100 = 10000
    - Can then retrieve required tuples
    - Counting number of matching tuples is even faster

位图索引（续）

- Bitmap indices generally very small compared with relation size
  - E.g. if record is 100 bytes, space for a single bitmap is 1/800 of space used by relation.
    - If the number of distinct attribute values is 8, bitmap is only 1% of relation size
- Deletion needs to be handled properly
  - Existence bitmap to note if there is a valid record at a record location
  - Needed for complementation
    - not(A=v): (NOT bitmap-A-v) AND ExistenceBitmap
- Should keep bitmaps for all values, even null value
  - To correctly handle SQL null semantics for NOT(A=v):
    - intersect above result with (NOT bitmap-A-Null)

目录

SQL中的索引定义

- **create an index**

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

  E.g., **create index** b_index **on** branch(branch_name)

  – Use create unique index to indirectly specify and enforce the condition that the search key is a candidate key
    - Not required if SQL unique integrity constraint is supported

- **Drop an index**

  drop index <index-name>

补充学习（索引相关）

- **商用数据库**
  - Oracle索引结构：B树索引，位图索引
    - 《Oracle索引技术》，人民邮电出版社
  - IBM DB2索引结构：B^+树
  - Microsoft SQL Server索引结构：B树
- **开源数据库**
  - MySQL索引：B-Tree(B^+Tree)、Hash索引
  - Postgre SQL, MySQL, Ingres r3, MaxDB, Firebird (InterBase), MongoDB, SQLite, CUBRID, Cayley(Graph)
- **NoSQL数据库**
  - HBase, Cassandra, MongoDB, Redis
  - OceanBase, openGauss, 人大金仓, X-DB, 达梦 …