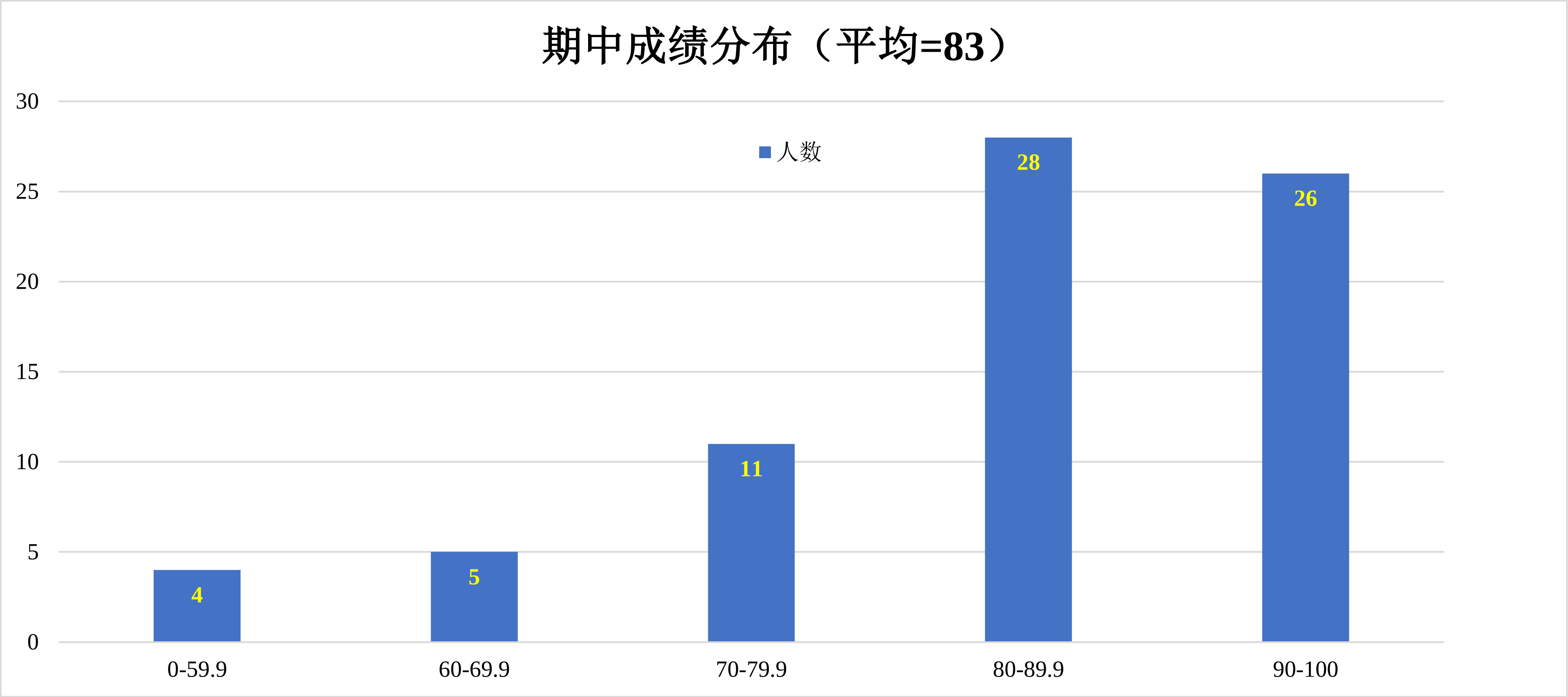
期中考试成绩统计


期中成绩分布（平均=83）
30
25
20
15
10
5
0
人数
4
5
11
28
26
0-59.9
60-69.9
70-79.9
80-89.9
90-100

Introduction to Databases
《数据库引论》

Lecture 8: Indexing & Hashing
第8讲：索引与哈希

周水庚 / Shuigeng Zhou

邮件: sgzhou@fudan.edu.cn 网址：admis.fudan.edu.cn/sgzhou

复旦大学计算机科学技术学院

Outline of the Course

- **Part 0: Overview**
  - Lect. 1 (Feb. 29) - Ch1: Introduction
- **Part 1 Relational Databases**
  - Lect. 2 (Mar. 7) - Ch2: Relational model (data model, relational algebra)
  - Lect. 3 (Mar. 14) - Ch3: SQL (Introduction)
  - Lect. 4 (Mar. 21) – Ch4/5: Intermediate and Advanced SQL
- **Part 2 Database Design**
  - Lect. 5 (Mar. 28) - Ch6: Database design based on E-R model
  - **Apr. 4 (Tomb-Sweeping Day): no course**
  - Lect. 6 (Apr. 11/18) - Ch7: Relational database design
- **Midterm exam: Apr. 25**
  - 13: 00-15: 00, H3109

☞ **Part 3 Data Storage & Indexing**
  - Lect. 7 (May 2 -> Apr. 28) - Ch12/13: Storage systems & structures
  - Lect. 8 (May 10) - Ch14: Indexing and Hashing
- **Part 4 Query Processing & Optimization**
  - Lect. 9 (May 17) - Ch15: Query processing
  - Lect. 10 (May 24) - Ch16: Query optimization
- **Part 5 Transaction Management**
  - Lect. 11 (May 31) - Ch17: Transactions
  - Lect. 12 (Jun. 7) - Ch18: Concurrency control
  - Lect. 13 (Jun. 14) - Ch19: Recovery system

Final exam: 13:00-15:00, Jun. 26

Outline

- ☞ Basic Concepts
- Ordered Indexing
- B^+-tree & B-tree Indices
- Static & Dynamic Hashing
- Ordered Indexing vs. Hashing
- Index Definition in SQL
- Multiple-key Access

Basic Concepts

- **Query （查询）**
  - The expression of user' requirements of data in the database using use some query language such as SQL
  - The major form of data access in DBs

- For example
  - ***select*** *loan_number*
    from *loan*
    where *branch_name = 'Perryridge'* ***and*** *amount > 1200*

- Indexes (索引) are a kind of data structures for speeding up query processing

Basic Concepts

- **Indexing mechanisms**
  - Speed up the access to desired data
  - Index files are typically much smaller than the original 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)**

search-key	pointer

- **Two kinds of indices**
  - **Ordered index (顺序索引):** search keys are stored in sorted order
  - **Hash index (散列索引):** search keys are distributed uniformly across "buckets" using a "hash function"

Why indexes work?

- 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Page”，一个页在磁盘上占用8K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/8K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/8K字节=**10**个页面中检索到满足条件的索引块（可以放在内存中），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

Index Evaluation Metrics

- **Access types** supported efficiently
  - Equal-query (等值查询), Range-query (范围查询), kNN……

```
select loan_number
from   loan
where  branch_name = 'Perryridge'
```

```
select loan_number
from   loan
where  amount > 1200
```

- **Access time**:访问时间
- **Update (maintenance) time**
  - **Insertion time**：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
  - **Deletion time**：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
- **Space overhead**：空间开销，一个索引结构占用的额外存储空间

Outline

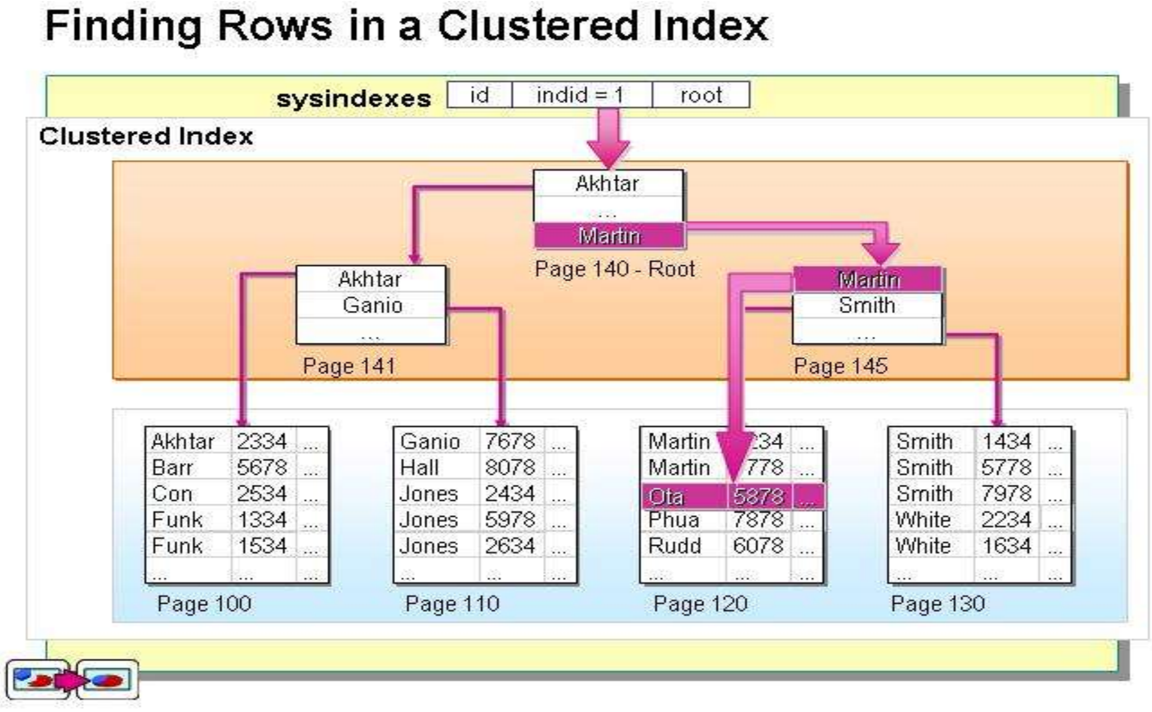
- Basic Concepts
- ☞ Ordered Indexing
- B^+-tree & B-tree Indices
- Static & Dynamic Hashing
- Ordered Indexing vs. Hashing
- Index Definition in SQL
- Multiple-key Access

Ordered Indexing-顺序索引

- **Ordered index**
  - Index entries are sorted on the search key value
  - Primary index and secondary index
  - **Primary index (主索引), clustering index 聚集索引**
    - 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 clustering index
  - **Secondary index (辅助索引), no-clustering index (非聚集索引)**
    - An index whose search key specifies an order different from the sequential order of the file
- **Index-sequential file (索引顺序文件)**
  - Ordered sequential file with a primary index
  - 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

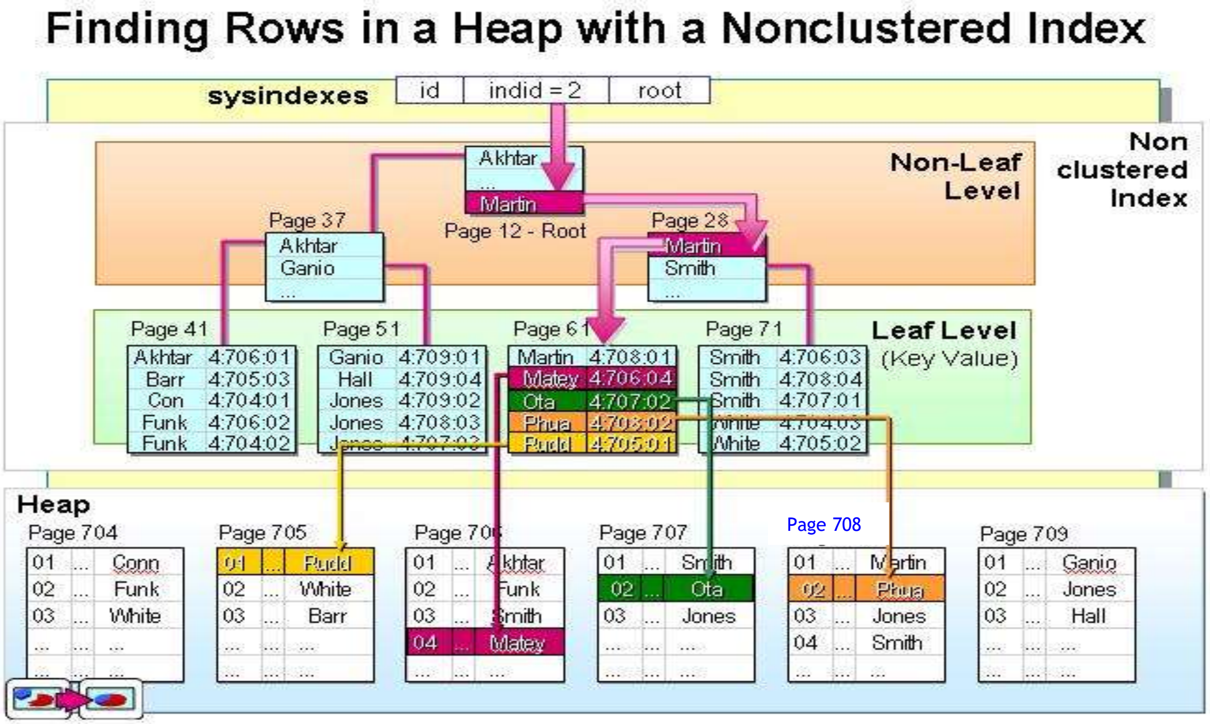
Primary Index: Clustering Index

- 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



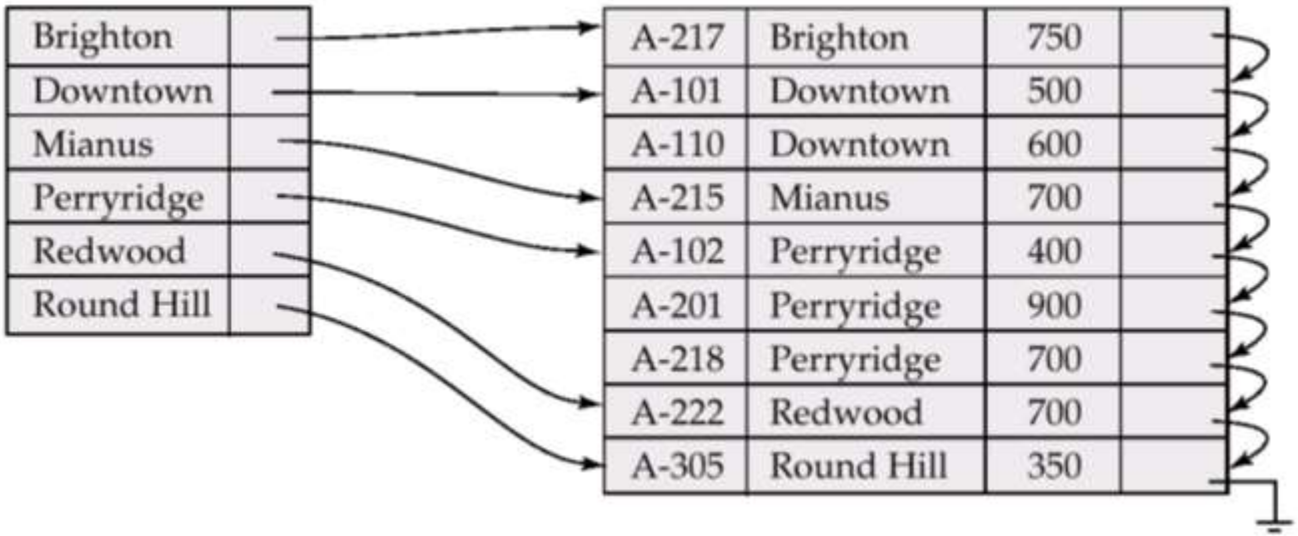
Secondary Index: Non-clustering Index

- 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



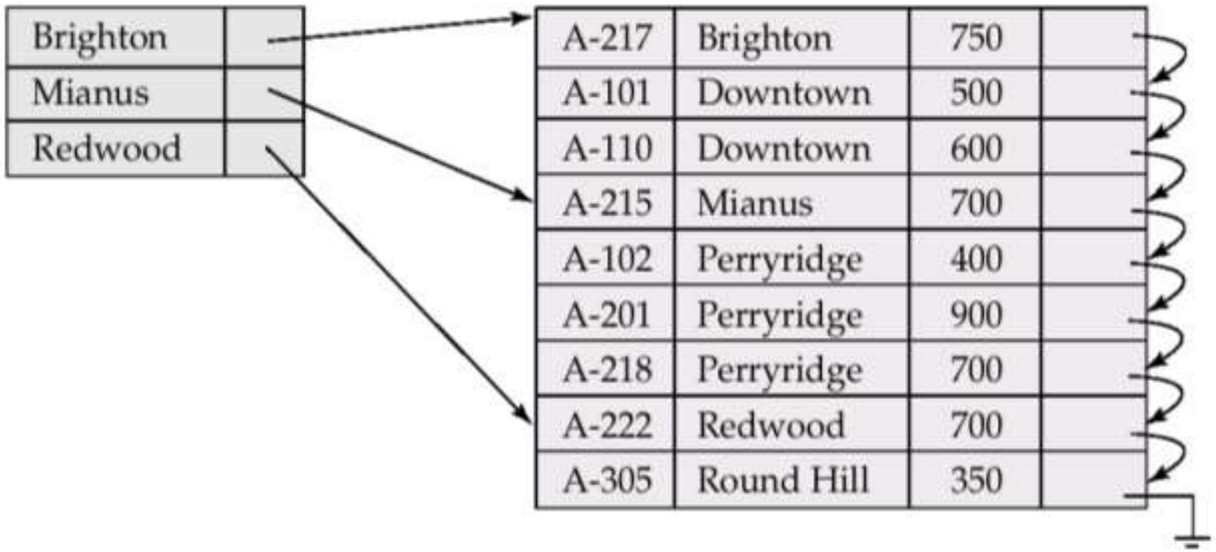
Dense Index

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



Sparse Index

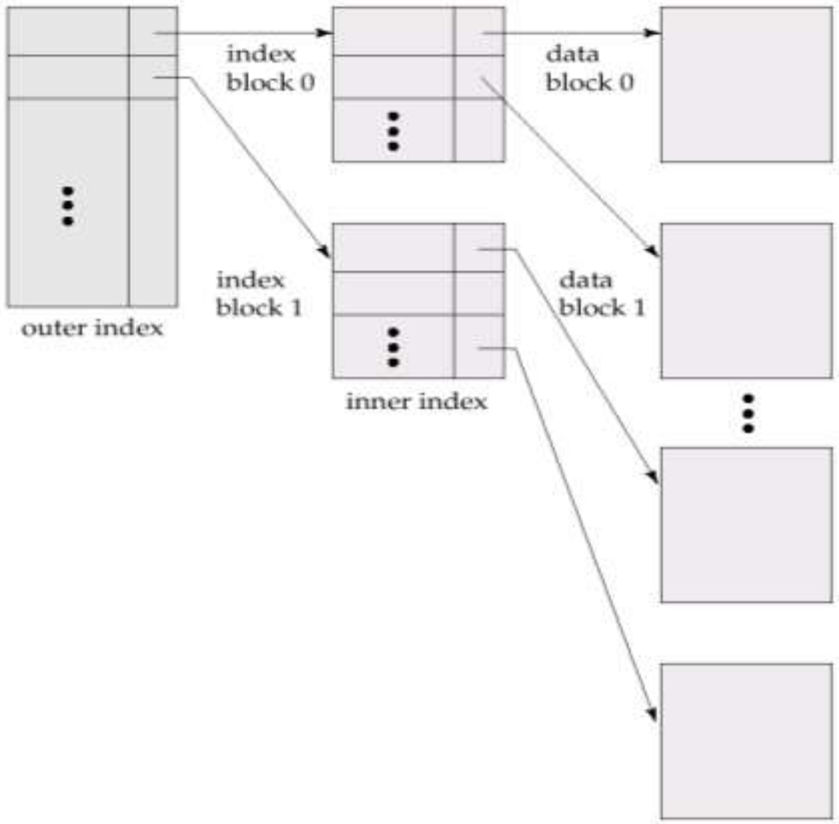
- **Sparse Index (稀疏索引)**
  - Contain index records for **only some search-key values** when records are sequentially ordered on search-key (**why?**)



Multilevel Index (多级索引)

- 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
  - **outer index** – a sparse index of primary index
  - **inner index** – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on

Multilevel Index (Cont.)

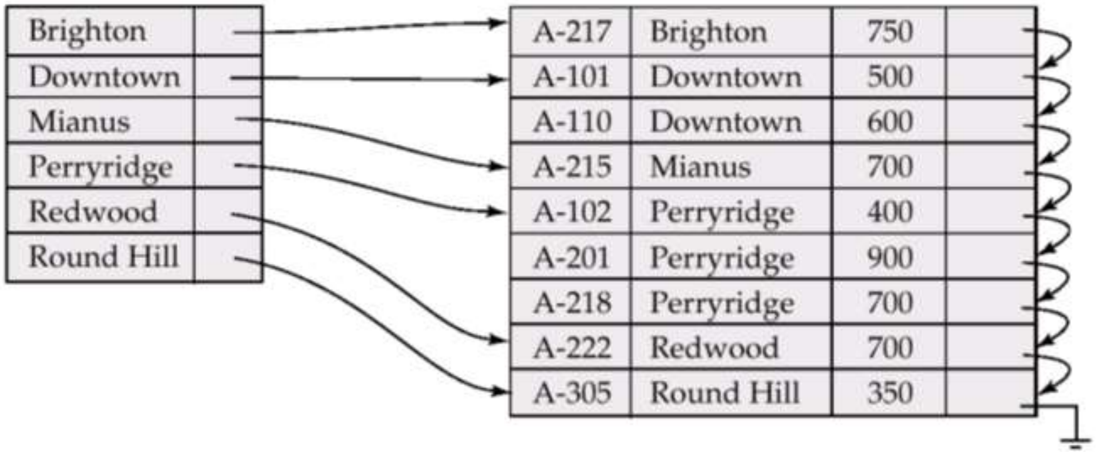


Dense vs. Sparse Index

- 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**

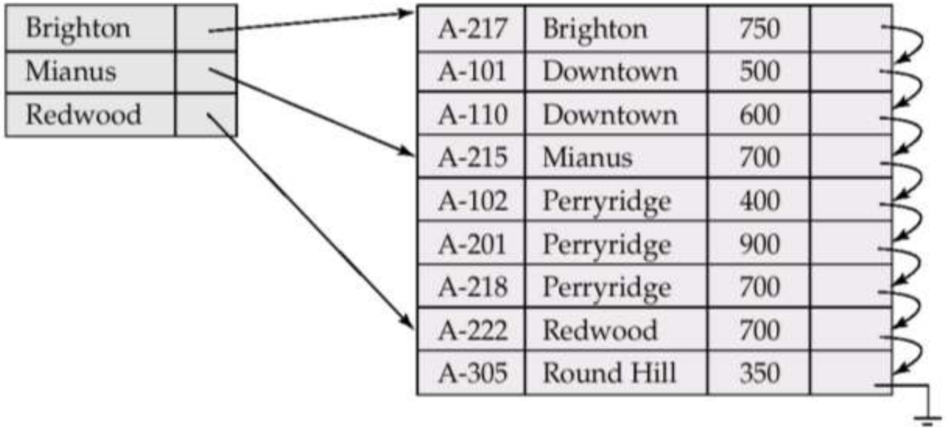
Index Update: Deletion

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



Index Update: Deletion

- **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



Index Update: Insertion

- **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

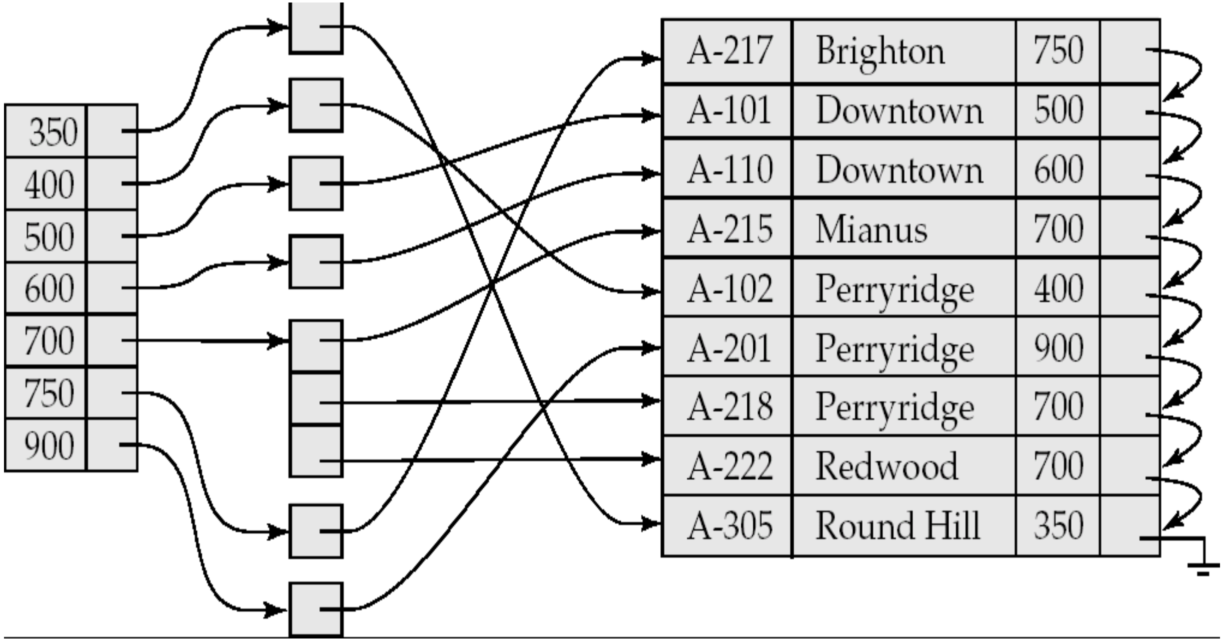
Dense vs. Sparse Index

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

Secondary Indices

- **Querying by secondary indices**
  - **Example 1:** In the account relation stored sequentially by account number, we may want to find all accounts in a particular branch
  - **Example 2:** to find all accounts with a specified balance or range of balances
- **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

Secondary Index on Balance Field of Account



bucket

Primary and Secondary Indices

- **Secondary indices have to be dense (why?)**
- When a file is modified, every 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**
  - each record access may fetch a new block from disk

Outline

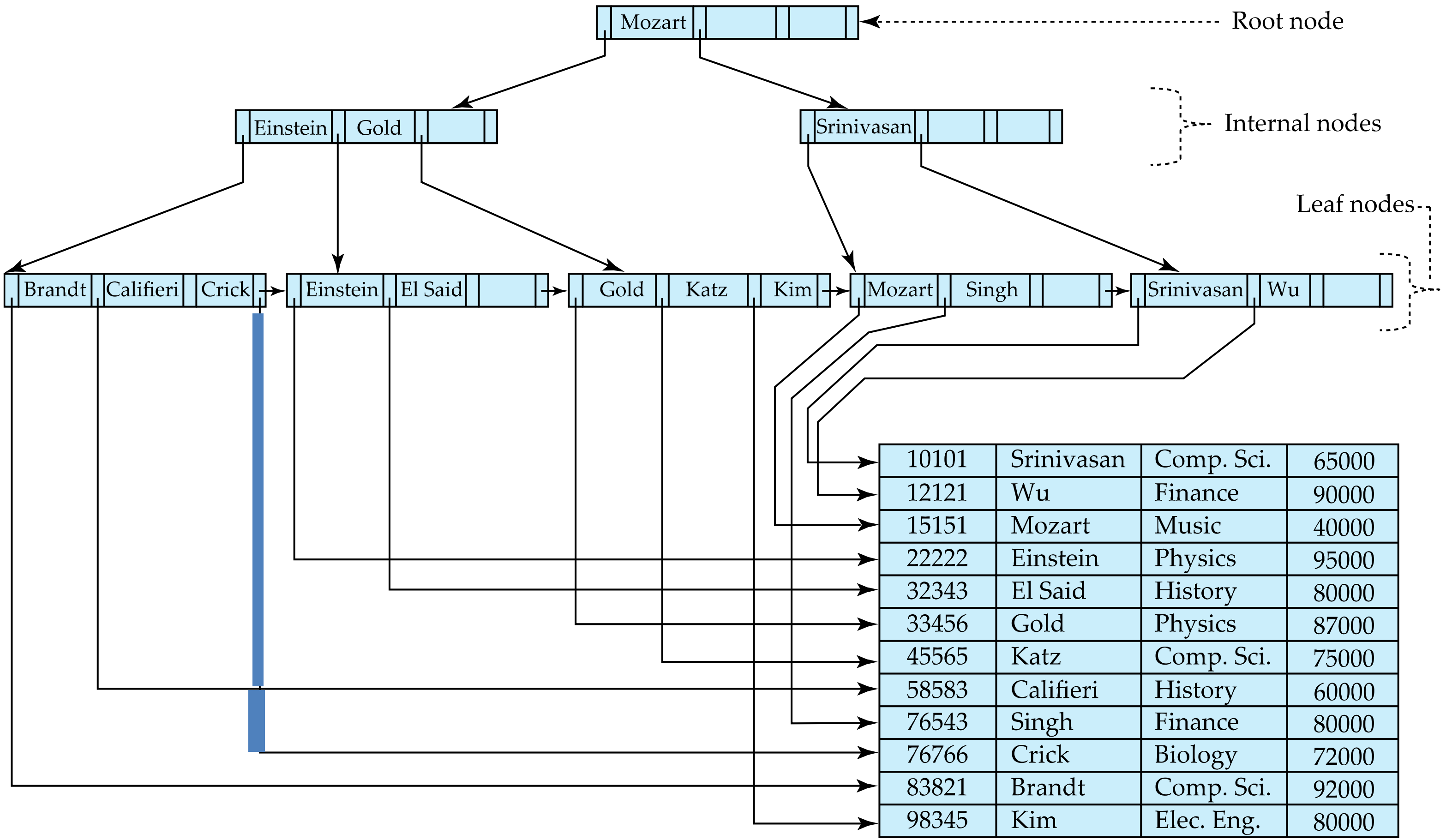
- Basic Concepts
- Ordered Indexing
- ☞ B^+-tree & B-tree Indices
- Static & Dynamic Hashing
- Ordered Indexing vs. Hashing
- Index Definition in SQL
- Multiple-key Access

B^+-Tree Index Files

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, **space** overhead
  - B^+-tree is used widely since its advantages outweight the disadvantages

Example of B+-Tree



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 Index Files (Cont.)

- **Typical B^+-tree node**

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

  - 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**)

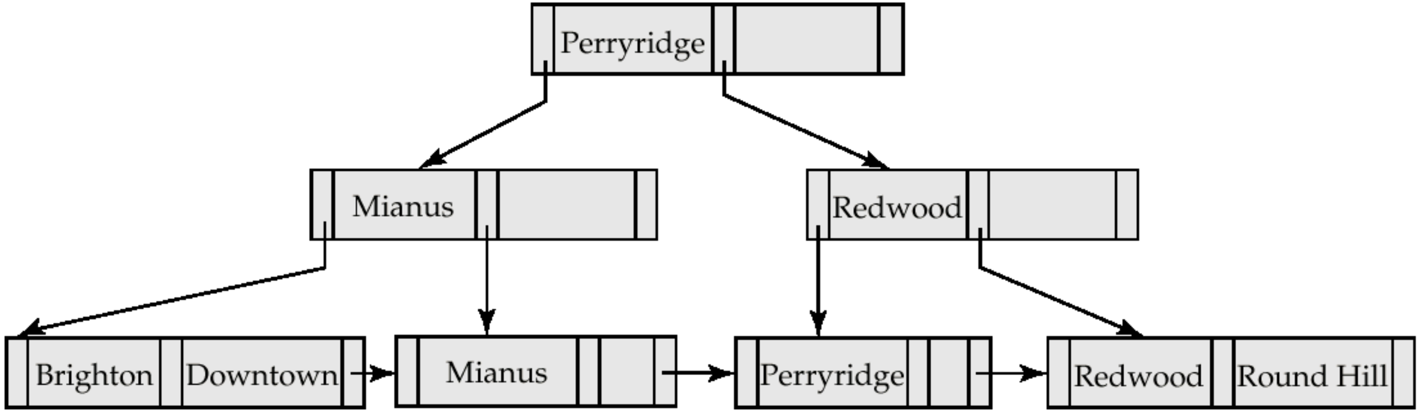
B^+-Tree Index Files (Cont.)

- 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**

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

    - 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**

Example of a B^+-tree



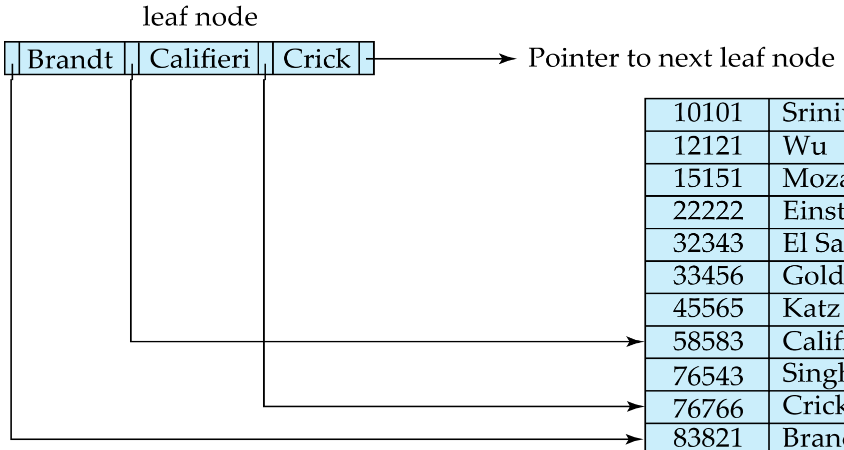
B^+-tree for *account* file (*n* = 3)

- 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

Leaf Node in B^+-Tree

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

- **Properties of a leaf node**
  - 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

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

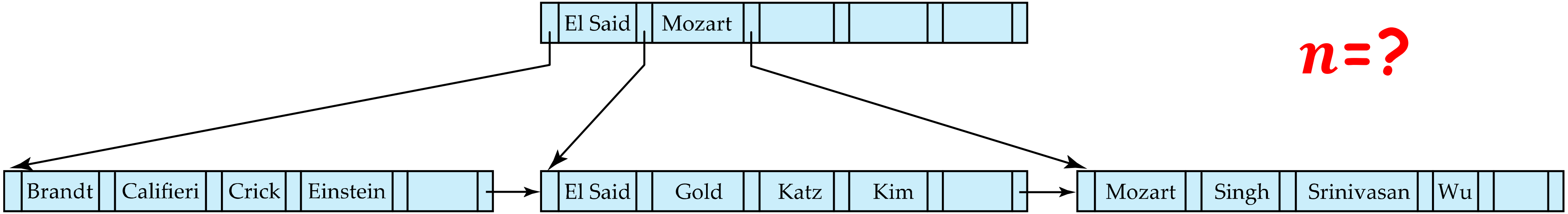
instructor file

Non-Leaf Nodes in B+-Tree

- **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}

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

Example of B+-tree



***n*=?**

- 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

Observations about B^+-tree

- 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

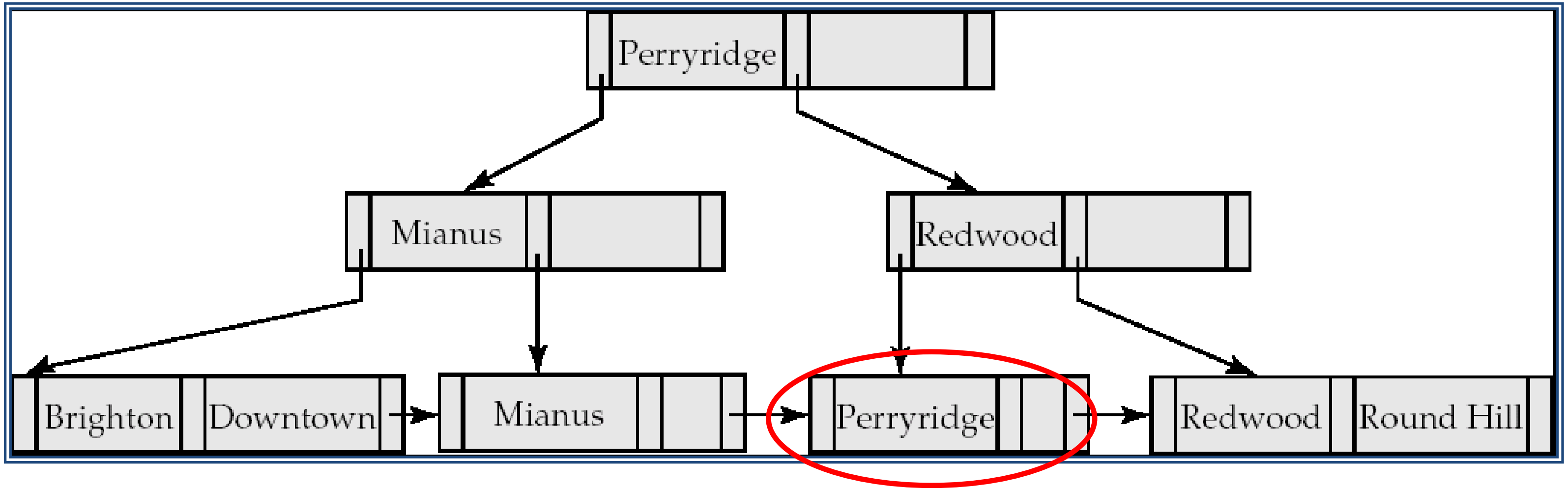
Queries on B+-Trees

- **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

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

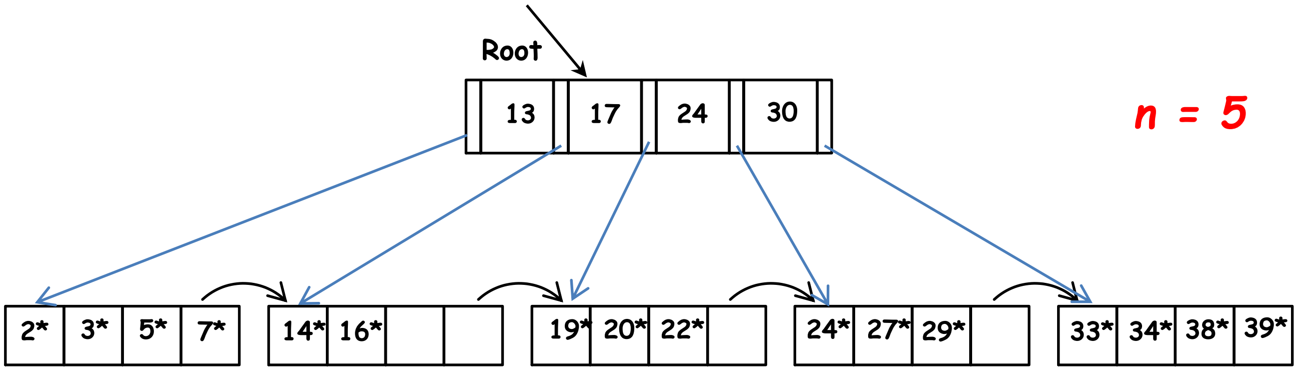
Example: Queries on B⁺-Tree

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



Example: Queries on B^+-Tree

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



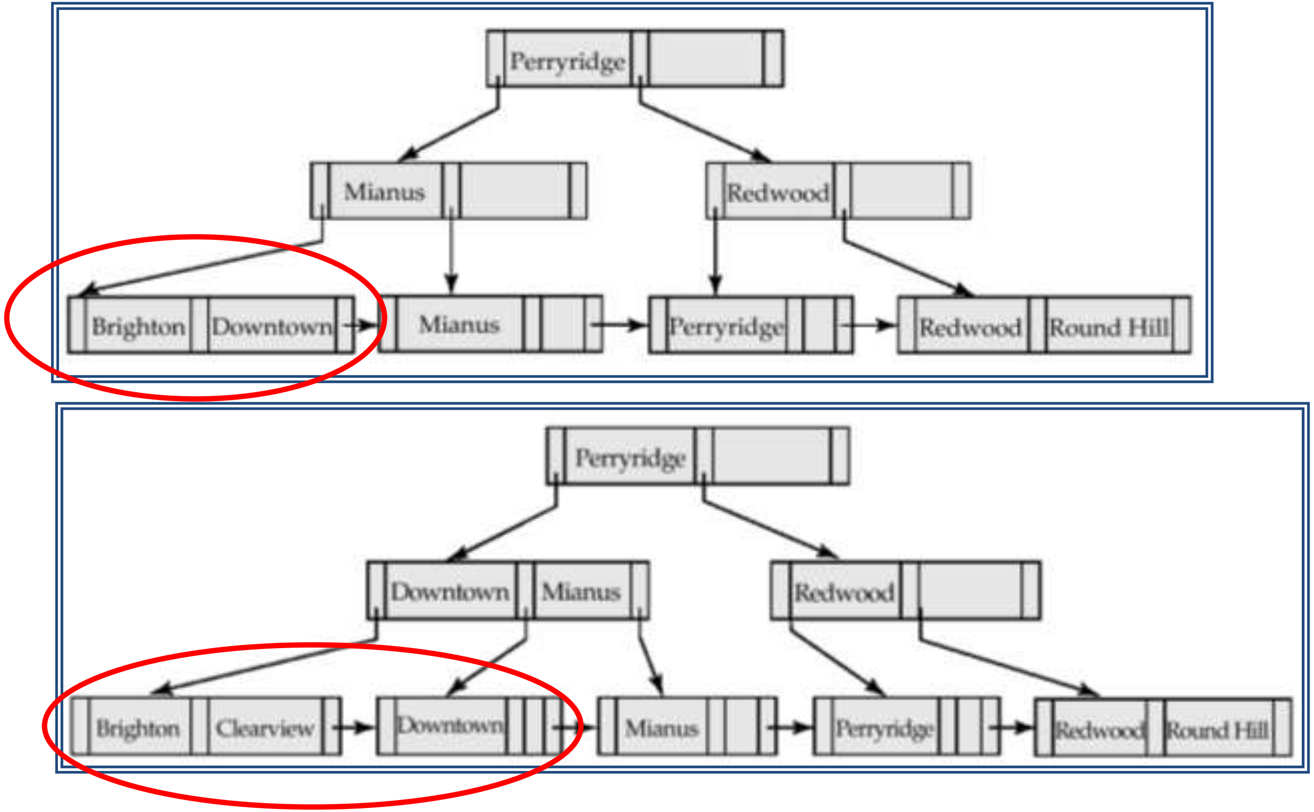
Queries on B⁺-Trees (Cont.)

- 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

Insertion in B^+-Tree

- **Find the leaf node in which the search-key value would appear**
  - If the search-key value is already in the leaf node
    - record is added to file
    - if necessary, a pointer is inserted into the bucket
  - If the search-key value is not in certain node, add the record to the main file and create a bucket if necessary. 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**

Insertion in B^+-Tree (Cont.)



n = 3

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

Insertion in B^+-Tree (Cont.)

- **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 **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

Insertion in B+-Tree (Cont.)

- **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

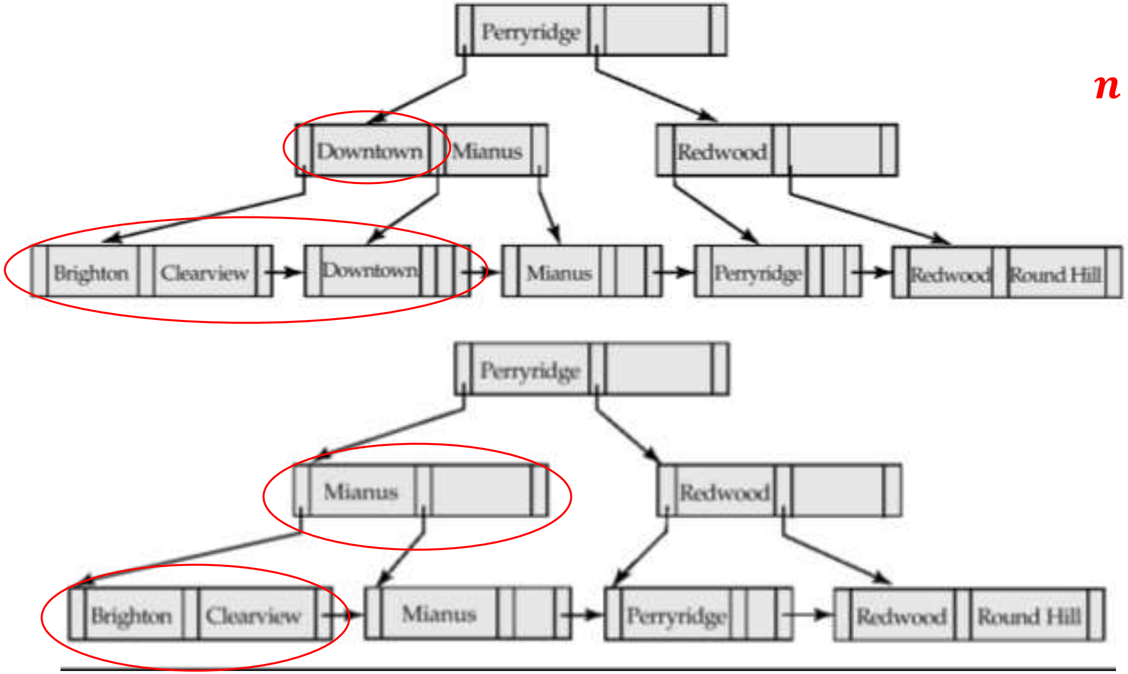
P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

Deletion in B^+-Tree

- 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

Examples of B⁺-Tree Deletion

Before and after deleting "Downtown"



***n* = 3**

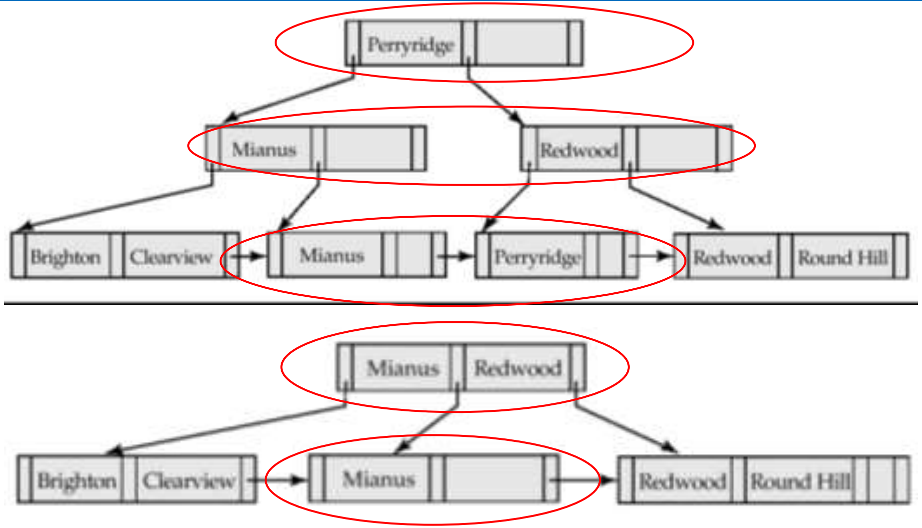
- Deleting "Downtown" causes **merging of under-full leaves**
- The removal of the leaf node containing "Downtown" did not result in its parent having too little pointers. So the cascaded deletions stopped with the deleted leaf node's parent

Deletion in B^+-Tree (Cont.)

- 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

Examples of B+-Tree Deletion (Cont.)

Deletion of "Perryridge"

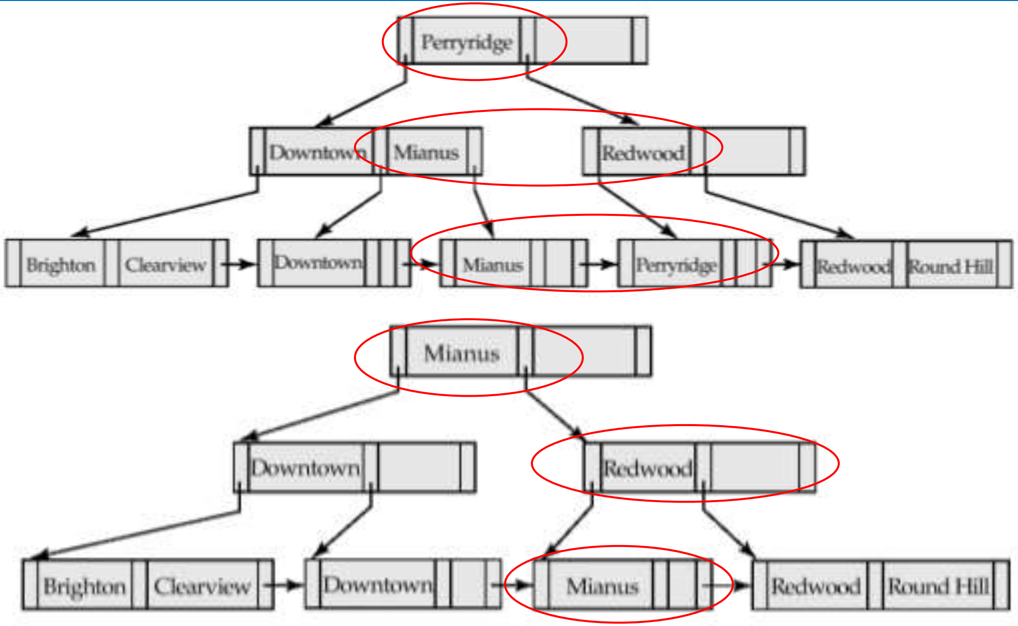


n = 3

- 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

Example of B+-tree Deletion (Cont.)

Deletion of "Perryridge"

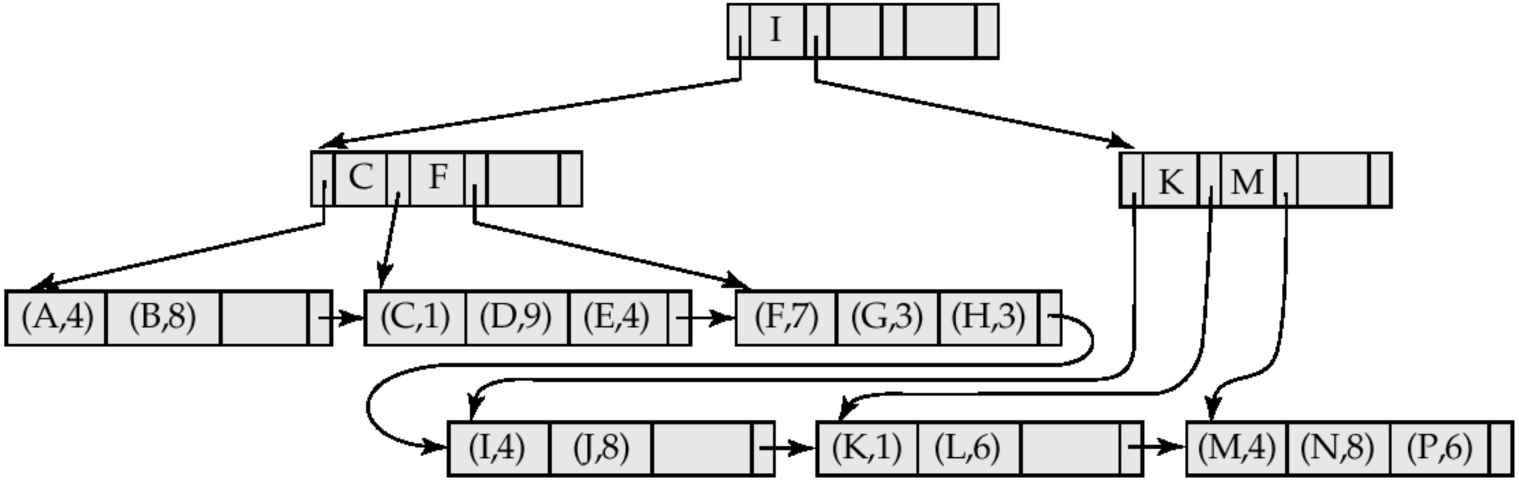


- Parent of leaf containing Perryridge became underfull, and borrowed a pointer from its left sibling
- Search-key value in the parent's parent changes as a result

B^+-Tree File Organization

- Index file degradation (性能下降) 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
- Insertion and deletion are handled in the same way as the insertion and deletion of entries in a B^+-tree index

B+-Tree File Organization (Cont.)

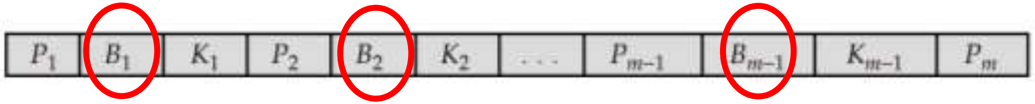


- Good space utilization is important since records use more space than pointers.
- To improve space utilization, involve more sibling nodes in redistribution
  - Involving 2 siblings or more in redistribution to avoid split / merge where possible

B-Tree Index Files

- 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**

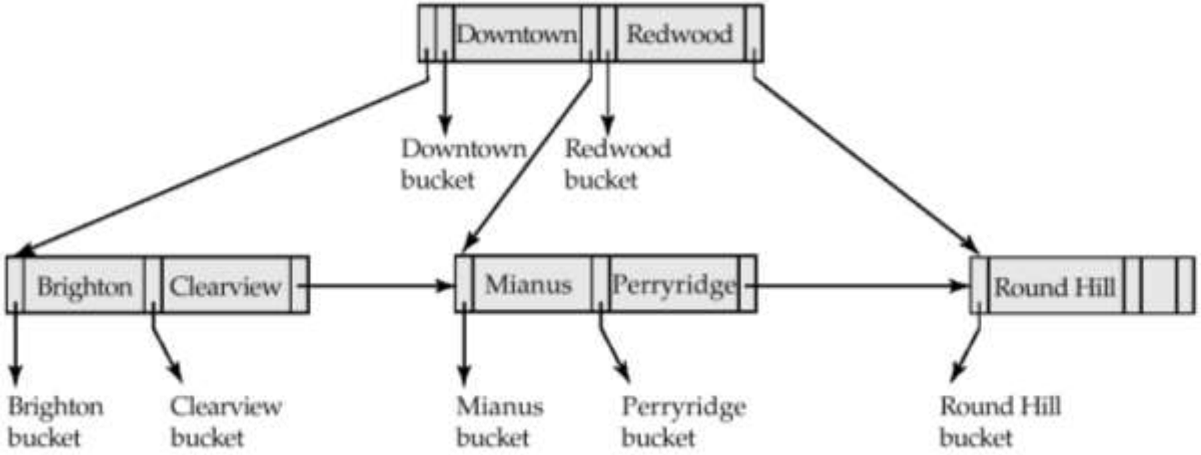
(a) *Leaf node*



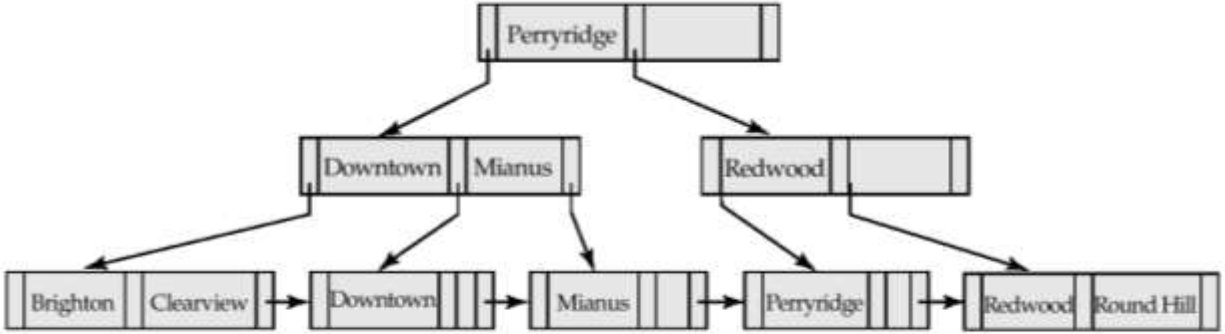
(b) *Non-leaf node*

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

B-Tree Index File



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



B-Tree Index Files (Cont.)

- **Advantages of B-Tree indices**
  - Use less tree nodes than B^+-Tree
  - Sometimes possible to find search-key value before reaching leaf node.
- **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

Outline

- Basic Concepts
- Ordered Indexing
- B^+-tree & B-tree Indices
- ☞ Static & Dynamic Hashing
- Ordered Indexing vs. Hashing
- Index Definition in SQL
- Multiple-key Access

Static Hashing

- 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

Example of Hash File Organization (Cont.)

- 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

Example of Hash File Organization

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

The binary representation of the ***i-th*** character is assumed to be the integer ***i***

h(Perryridge) = 125 mod 10 = **5**
h(Round Hill) = 113 mod 10 = **3**
h(Brighton) = 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

Hash Functions

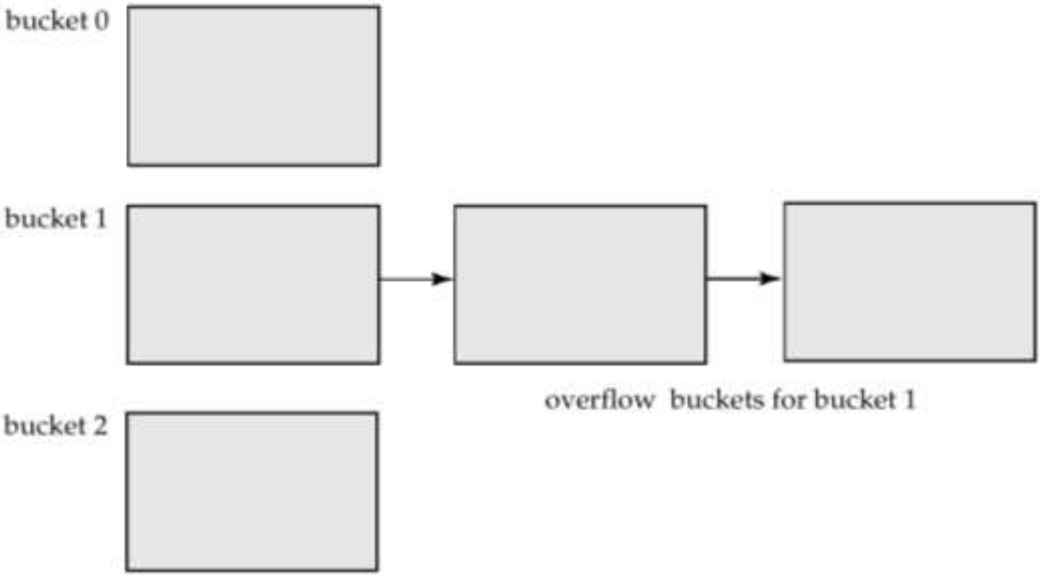
- Worst hash function maps all search-key values to the same bucket
- An ideal hash function is uniform, i.e., each bucket is assigned the same number of search-key values from the set of all possible values
- Ideal hash function is random, so each bucket will have the same number of records assigned to it irrespective of the actual distribution of search-key values in the file
- Typical hash functions perform computation on the internal binary representation of the search-key

Handling of Bucket Overflows

- **Bucket overflow** can occur because of
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to two reasons:
    - multiple records have same search-key value
    - chosen 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**

Handling of Bucket Overflows (Cont.)

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



Hash Indices

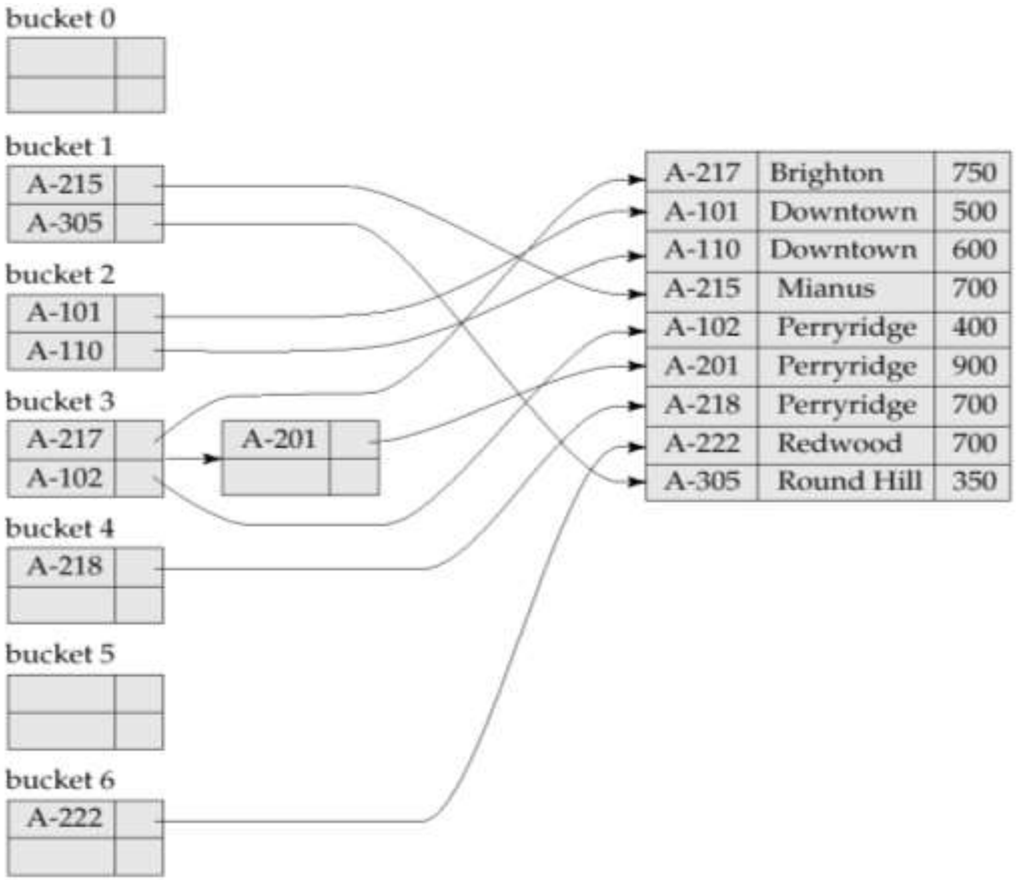
- **Hashing** can be used not only for **file organization**, but also for **index-structure** creation
- **A hash index** organizes the search keys, with their associated record pointers, into a hash file structure
- Strictly speaking, **hash indices are always secondary indices**
  - if the file itself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

Example of Hash Index

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.



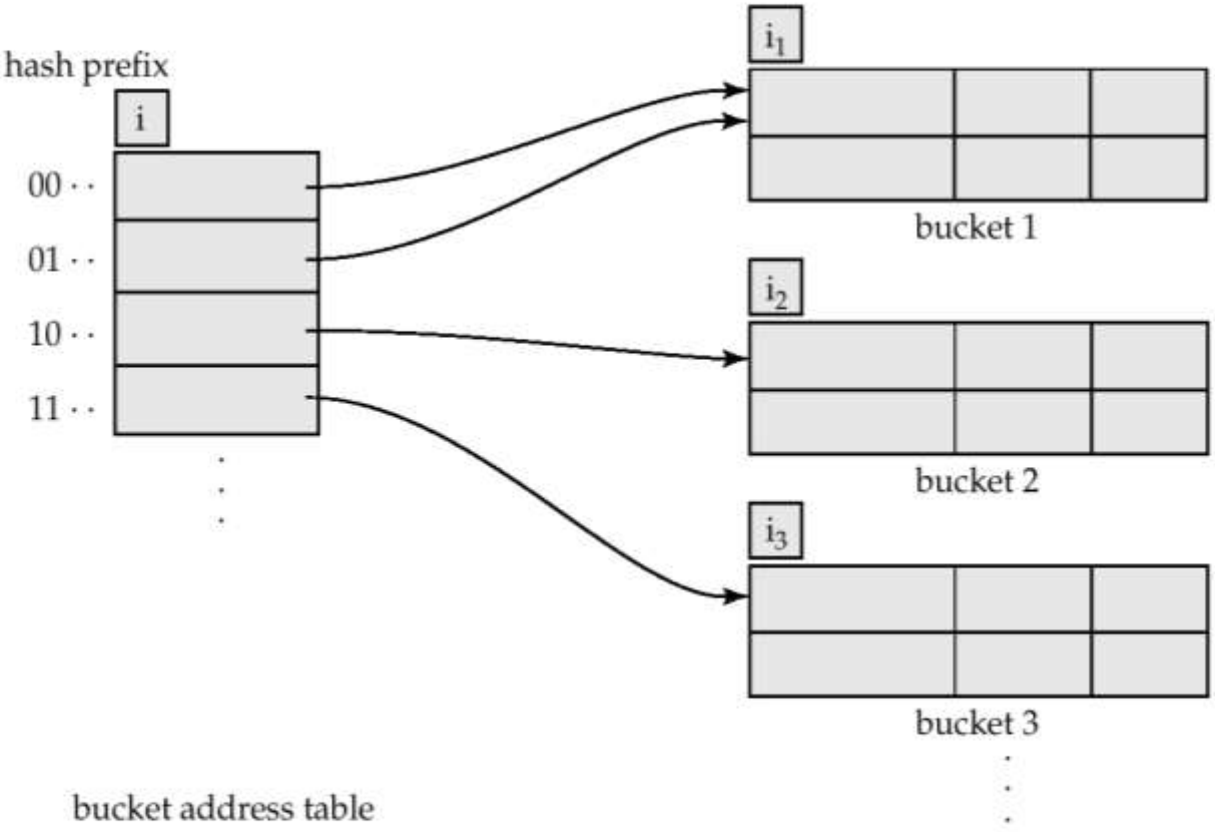
Deficiencies of Static Hashing

- 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**

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

General Extendable Hash Structure



Use of Extendable Hash Structure

- 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)

Updates in Extendable Hash Structure

- 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.

Updates in Extendable Hash Structure (Cont.)

- **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

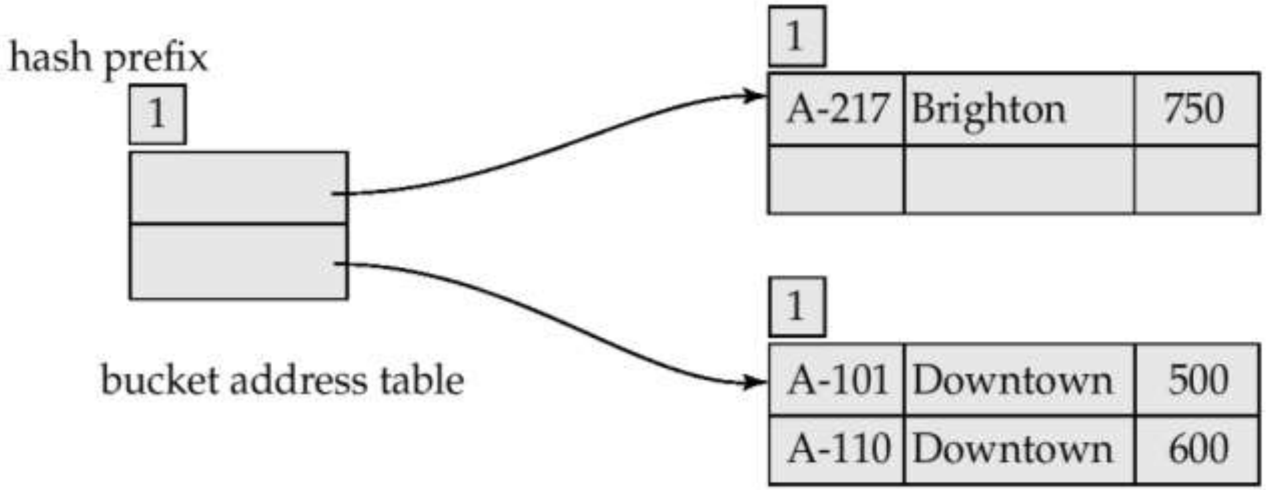
Use of Extendable Hash Structure: Example

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

Example (Cont.)

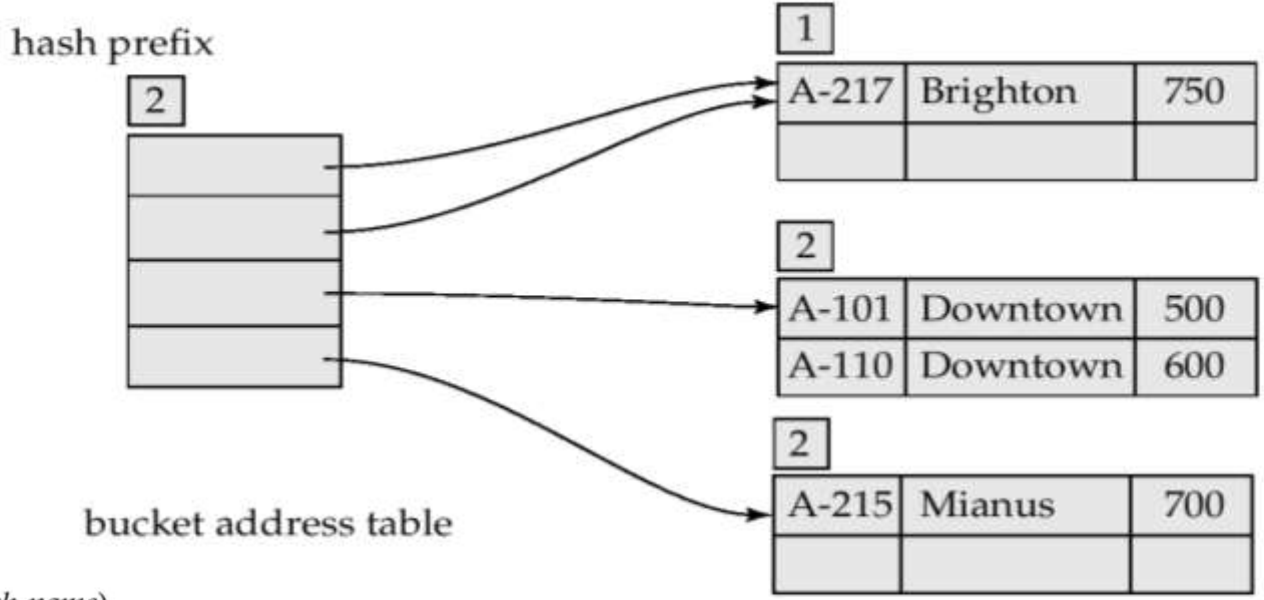
- Hash structure after insertion of one Brighton and two Downtown records



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

Example (Cont.)

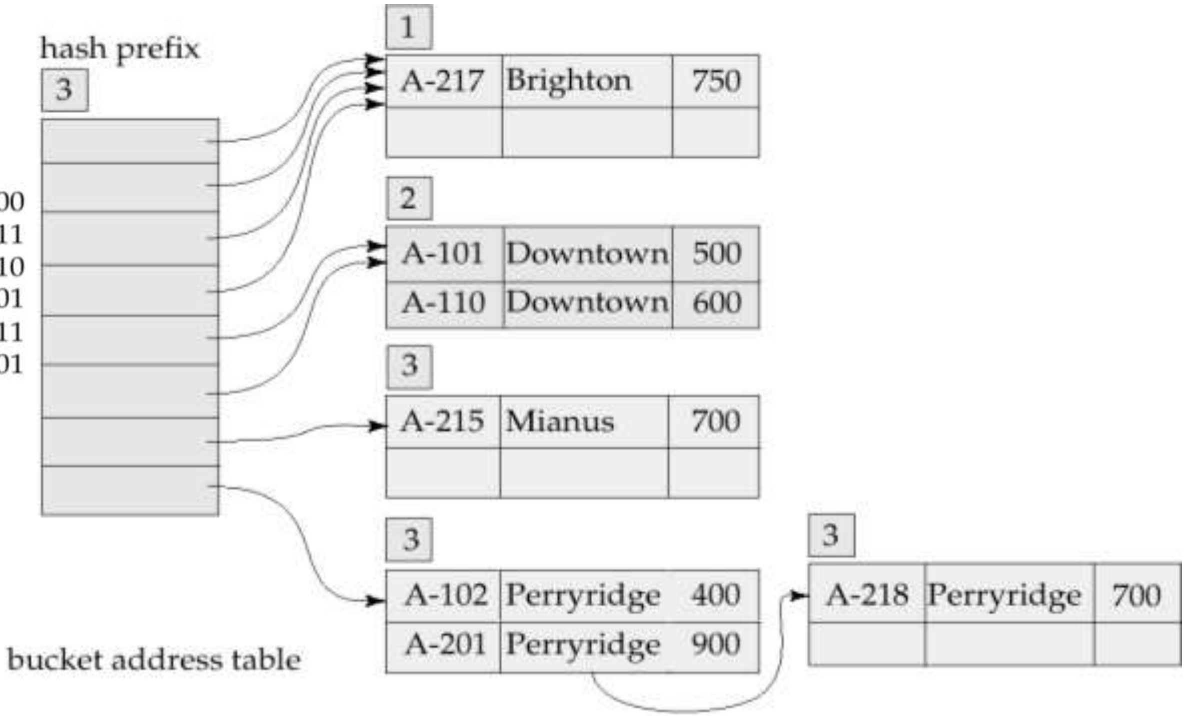
- Hash structure after insertion of Mianus record



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

Example (Cont.)

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

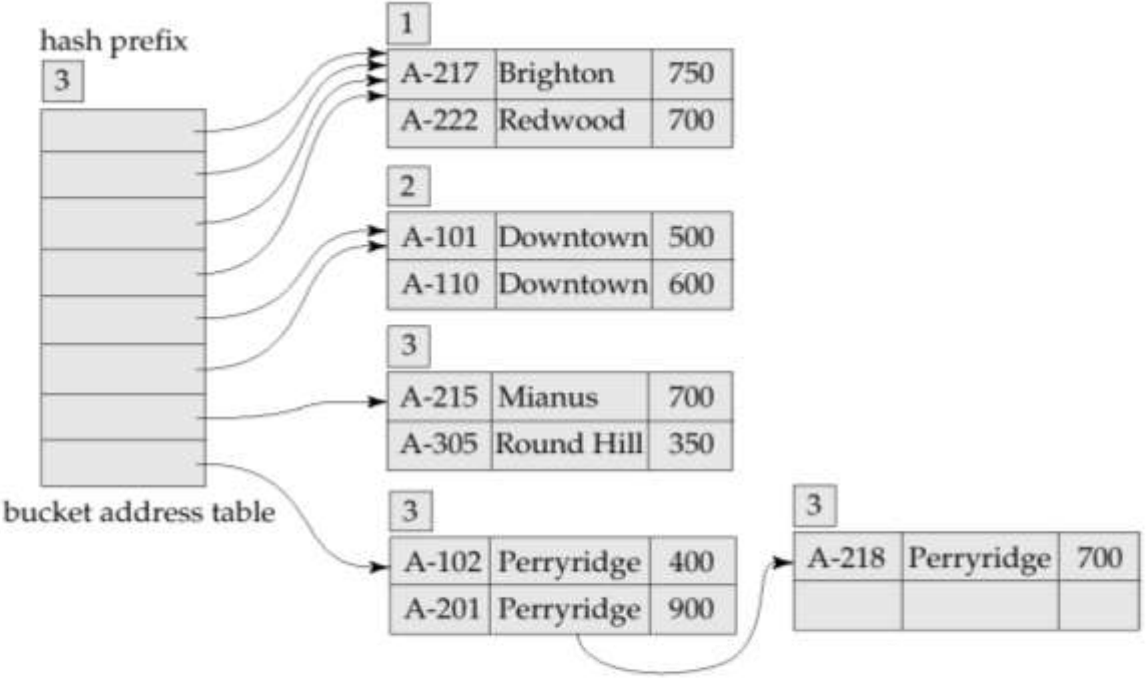


Hash structure after insertion of three Perryridge records

Example (Cont.)

- 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



Extendable Hashing vs. Other Schemes

- 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

Outline

- Basic Concepts
- Ordered Indexing
- B^+-tree & B-tree Indices
- Static & Dynamic Hashing
- ☞ Ordered Indexing vs. Hashing
- Index Definition in SQL
- Multiple-key Access

What to Consider for Index Selection?

- 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

Outline

- Basic Concepts
- Ordered Indexing
- B^+-tree & B-tree Indices
- Static & Dynamic Hashing
- Ordered Indexing vs. Hashing
- ☞ Index Definition in SQL
- Multiple-key Access

Index Definition in SQL

- **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 really required if SQL unique integrity constraint is supported

- **Drop an index**

  drop index <index-name>

Outline

- Basic Concepts
- Ordered Indexing
- B^+-tree & B-tree Indices
- Static & Dynamic Hashing
- Ordered Indexing vs. Hashing
- Index Definition in SQL

☞ Multiple-key Access

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

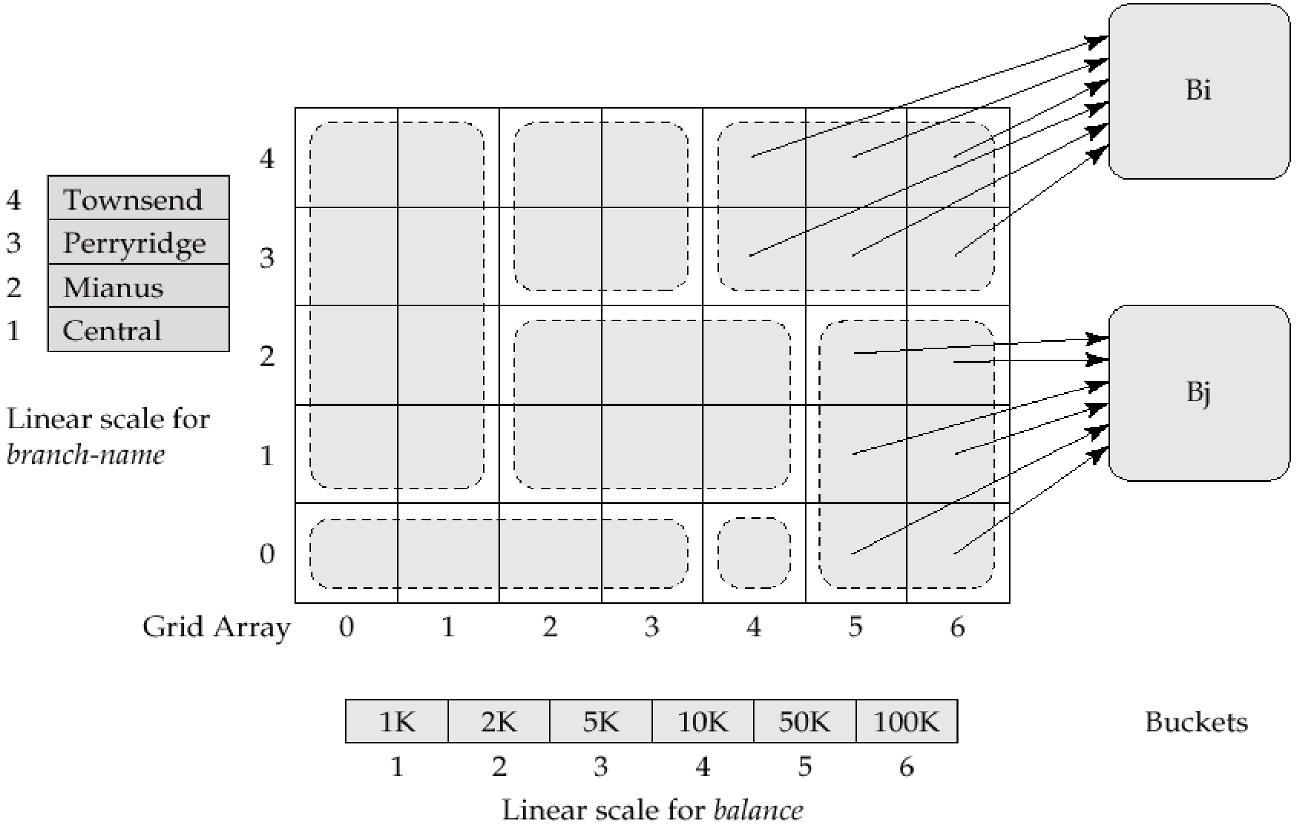
Indices on Multiple Attributes

- 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

Grid Files

- Structure used to speed up the processing of general **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

Example Grid File for account



Queries on a Grid File

- 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

Grid Files (Cont.)

- 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.

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

Bitmap Indices (Cont.)

- 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.)

- **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 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

Bitmap Indices (Cont.)

- 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 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)

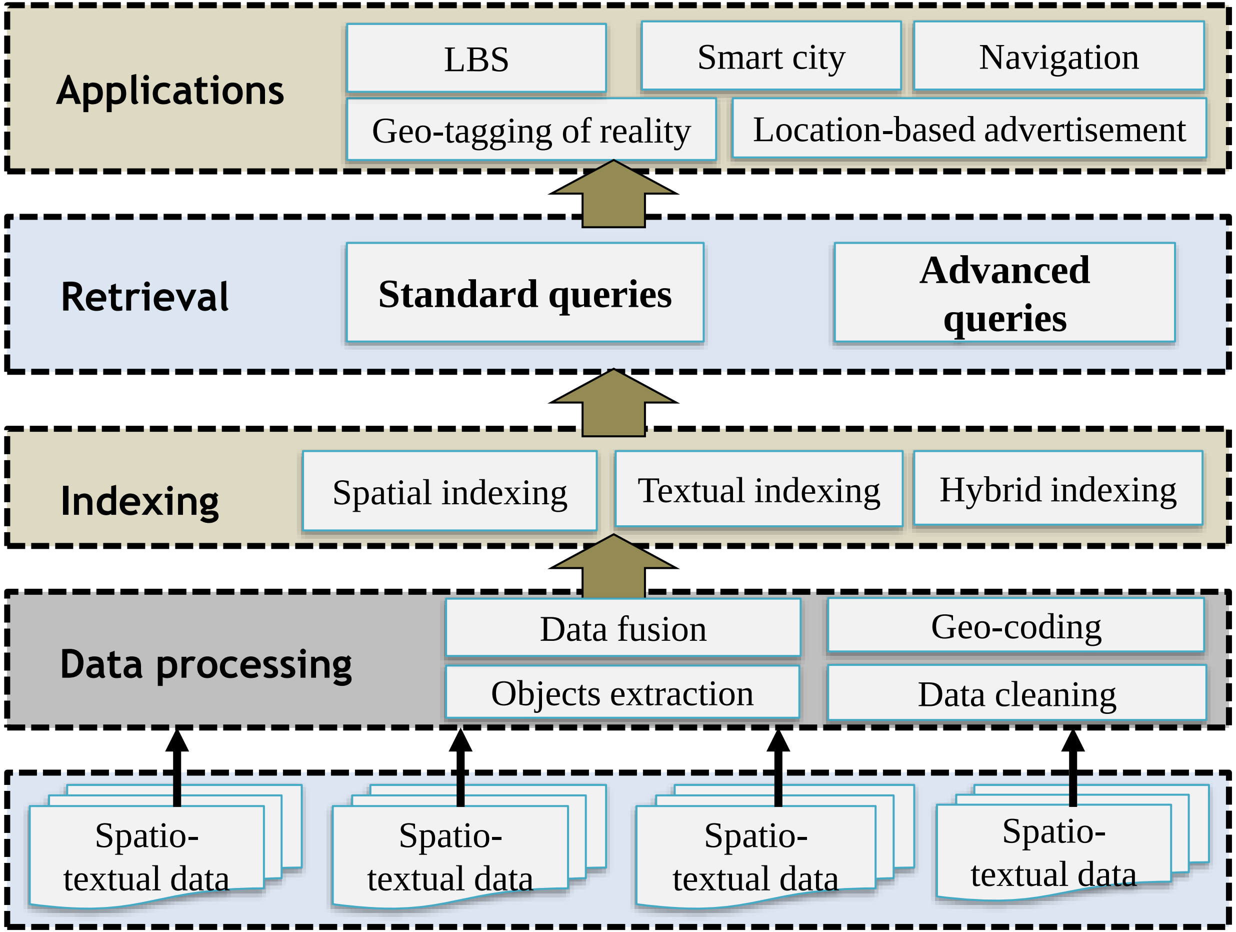
Assignments-Quiz

- **Q1**: Construct a B^+-tree from an empty tree. Each node can hold four pointers
  - The sequential values to be inserted are: 10, 7, 12, 5, 9, 15, 30, 23, 17, 26
  - Then delete 9, 10, 15, respectively
  - Please give the B^+ trees after each insertion and each deletion
- **Q2**: Compare B^+-tree and B-tree and describe their difference

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 ……

Research framework



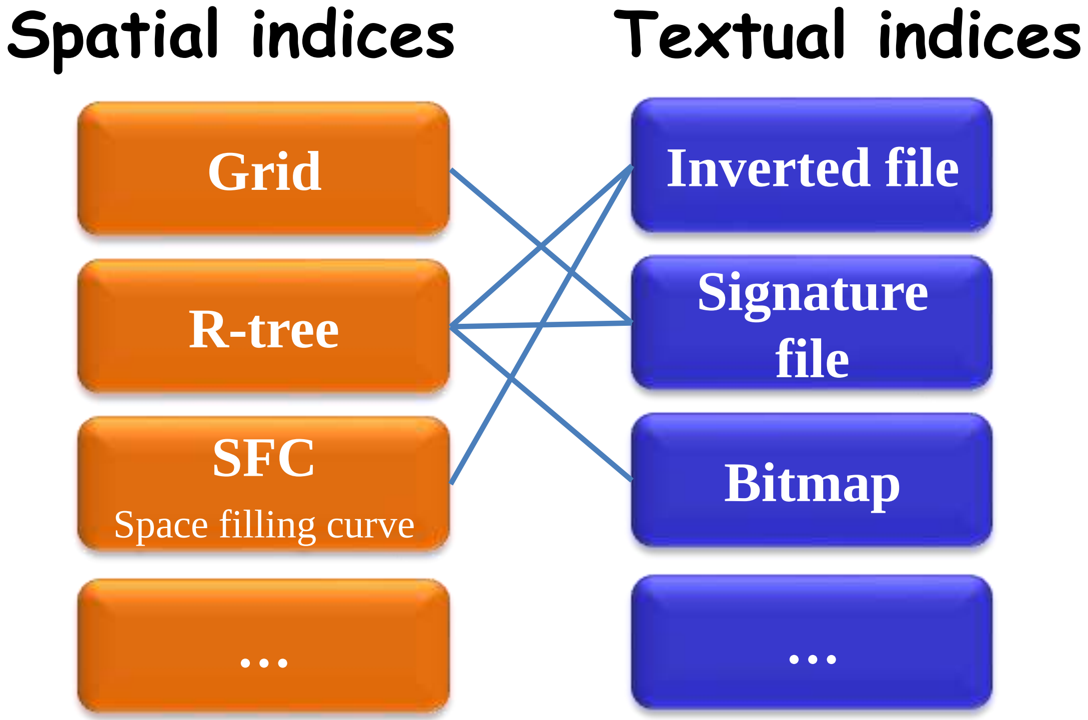
Spatio-textual objects

- $\boldsymbol{o}=(\boldsymbol{l},\boldsymbol{d})$
  - $o.l$: spatial location, $o.d$: text description

POI: shop, bank, restaurant, museum, school, hospital, etc.

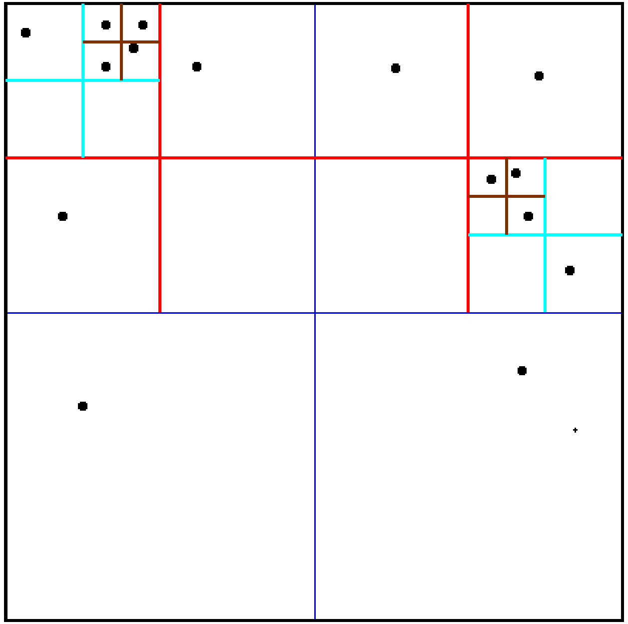
Geo-tagged web contents: news, images, videos, comments, micro-blogs

Spatio-textual indices

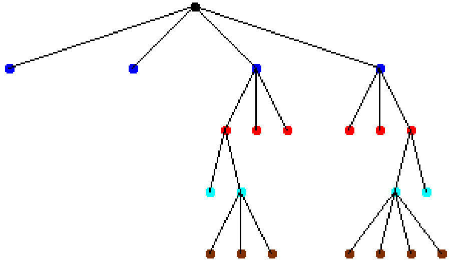


Spatial index: Grid index

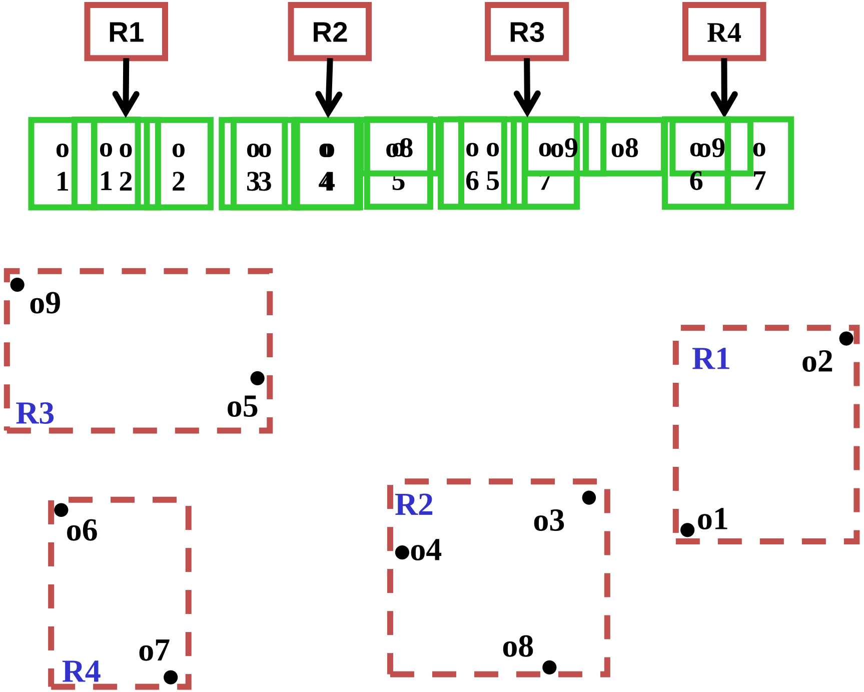
Partition



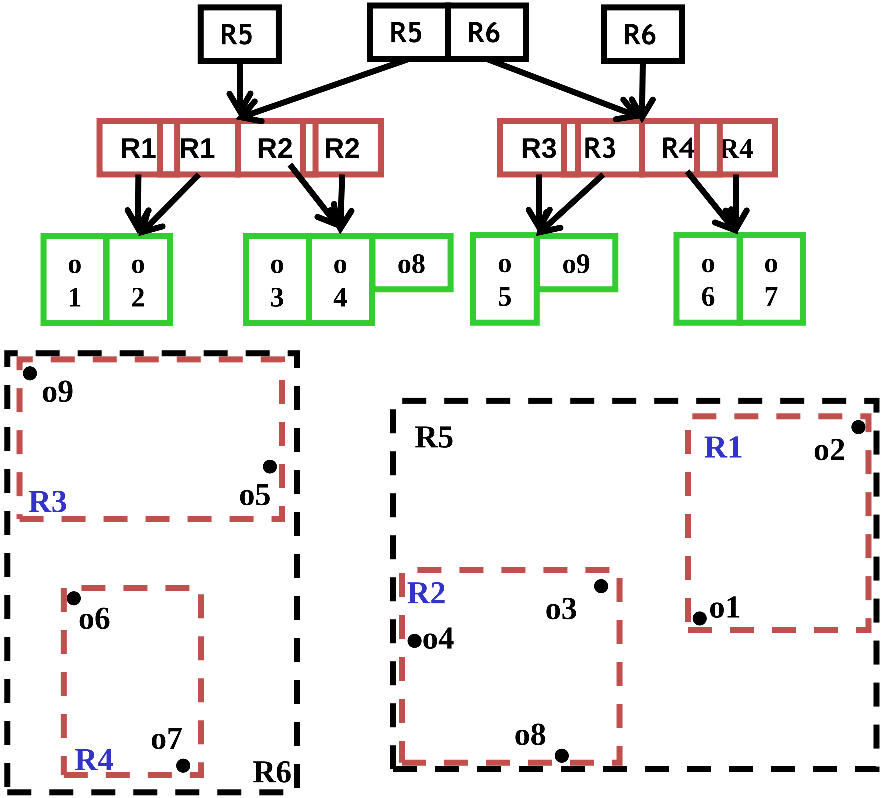
Tree structure



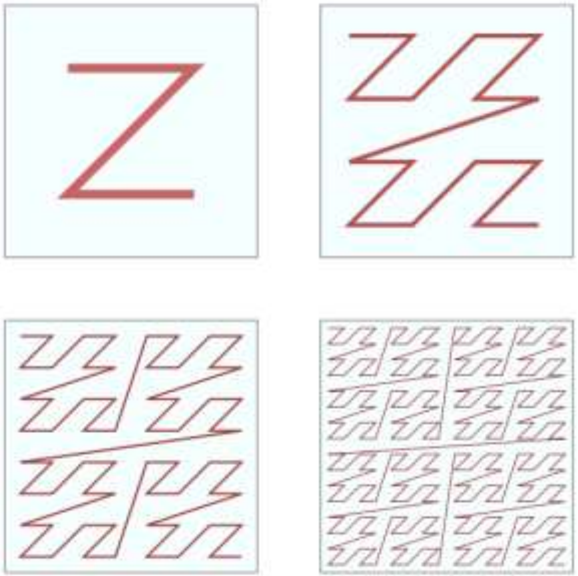
Spatial index: R-tree



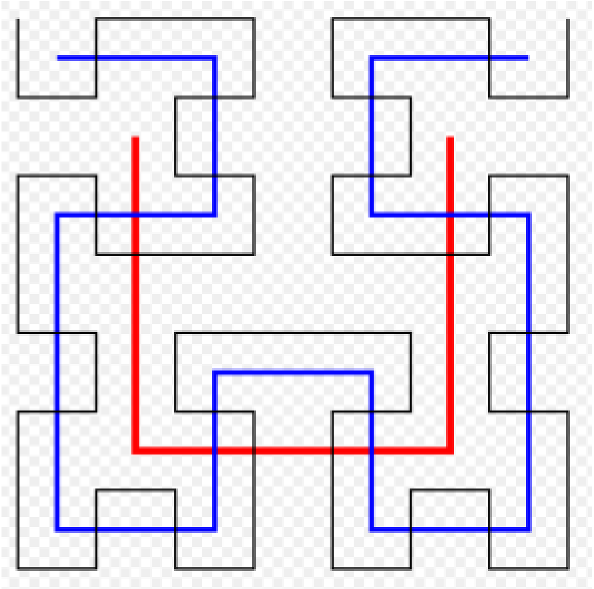
Spatial index: R-tree



Spatial index: space filling curve (SFC)

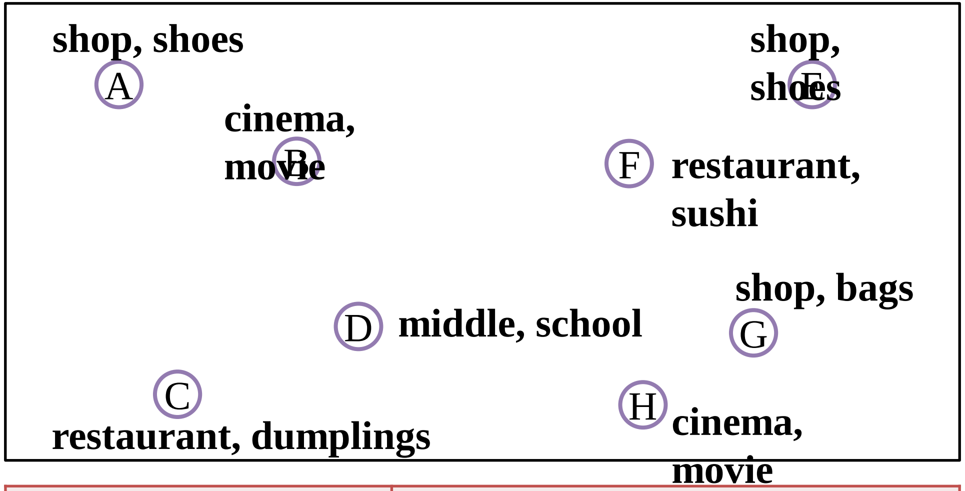


Z-curve



Hilbert curve

Textual index: inverted index



Spatio-textual objects

Keywords	Spatial-textual objects
shop	A, E, G
shoes	A, E
cinema	B, H
movie	B, H
restaurant	C, F
…	…

Inverted Index

Textual index: bitmap

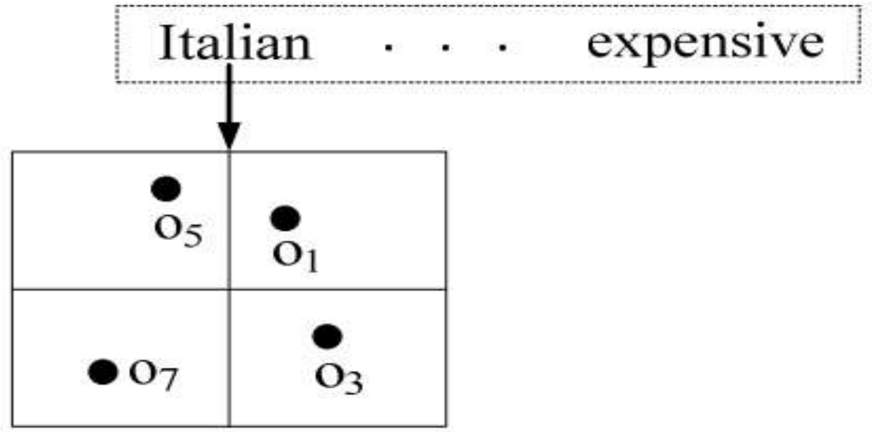
	k_1	k_2	k_3	k_4	k_5
$k_1k_2k_3$	1	1	1	0	0
$k_2k_4k_5$	0	1	0	1	1
k_2k_4	0	1	0	1	0
$k_1k_2k_4k_5$	1	1	0	1	1
k_4k_5	0	0	0	1	1
…					

Textual index: signature file

Terms/documents	Signature
k_1	sig(k_1)=0000000001
k_2	sig(k_2)=0000000010
k_3	sig(k_3)=1000000011
$k_1 k_2$	sig($k_1 k_2$)=sig(k_1) ∨ sig(k_2)=0000000011
$k_2 k_3$	sig($k_2 k_3$)=sig(k_2) ∨ sig(k_3)=1000000011
…	

ST index& TS index

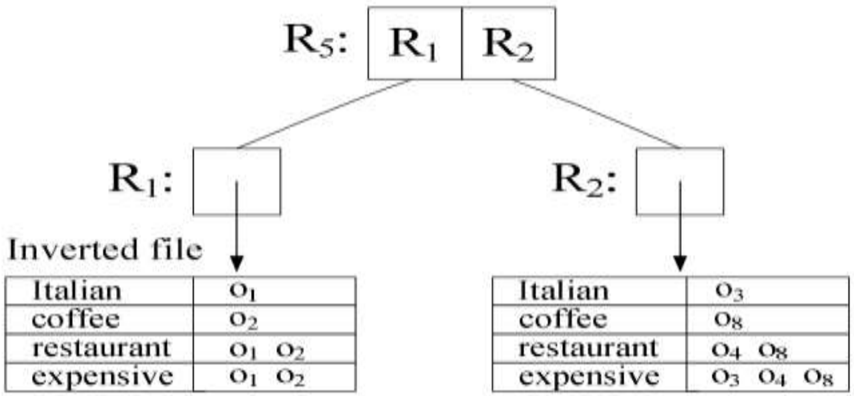
- **Grid index + Inverted file**
  - ST: spatial textual index (grid index first)
  - TS: textual spatial index (inverted file first)



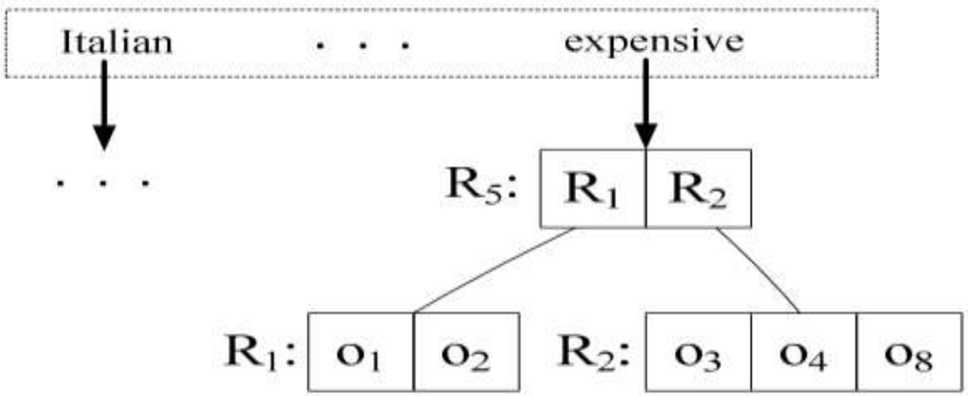
R*-tree-IF and IF-R*-tree

- **R*-tree + Inverted file**
  - R*-tree: a variant of R-tree

R*-Tree-IF



IF-R*-Tree



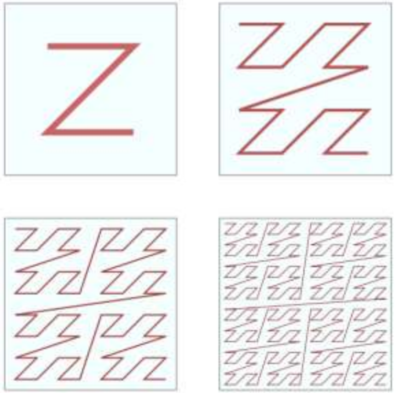
KR*-tree (Keyword R*-tree)

- **R*-tree + Inverted file**
  - Each node is virtually augmented with the set of keywords that appear in its subtree.
  - Nodes are organized into inverted file

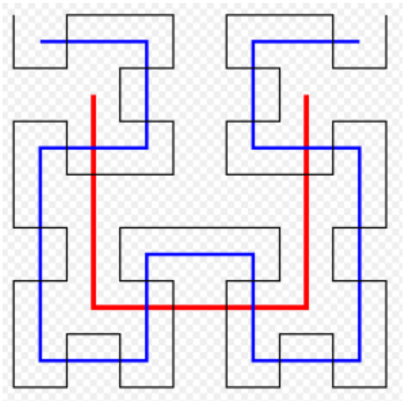
Keyword	**Tree nodes**
Italian	$R_1, R_2, R_3, R_4, R_5, R_6$
coffee	R_1, R_2, R_4, R_5, R_6
restaurant	$R_1, R_2, R_3, R_4, R_5, R_6$
Pizza	R_2, R_4, R_5, R_6
Expensive	R_1, R_2, R_5

SFC-Quad

- **Inverted file + Filling curve**
  - **Inverted file + Hilbert curve:** inverted lists are laid out along a Hilbert curve on disk.
  - **Inverted file + Z-curve:** the objects in each inverted list are assigned and ordered based on their spatial positions on the Z-curve.



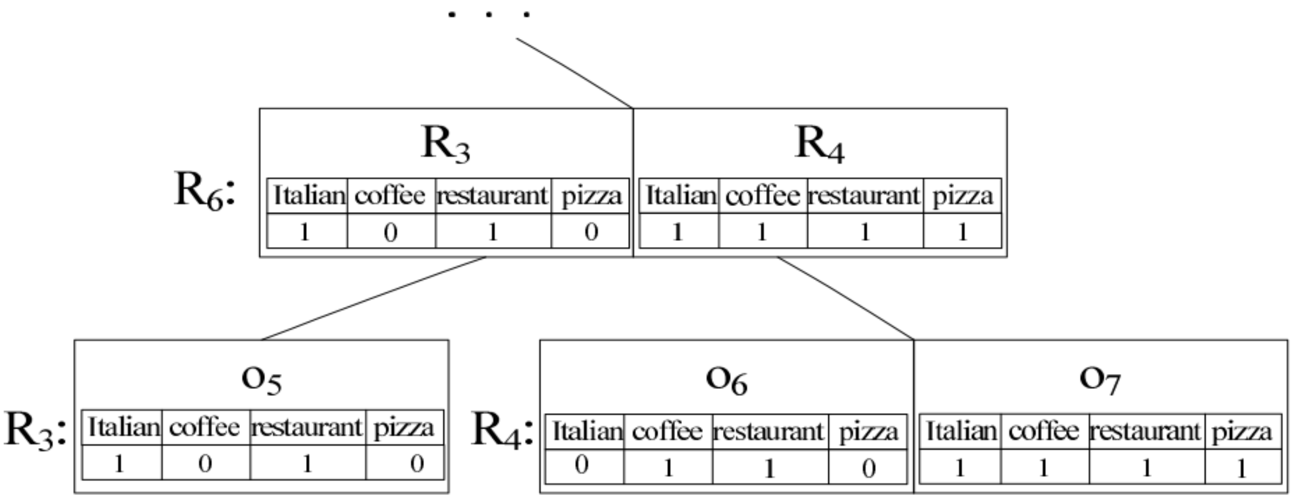
Z-curve



Hilbert curve

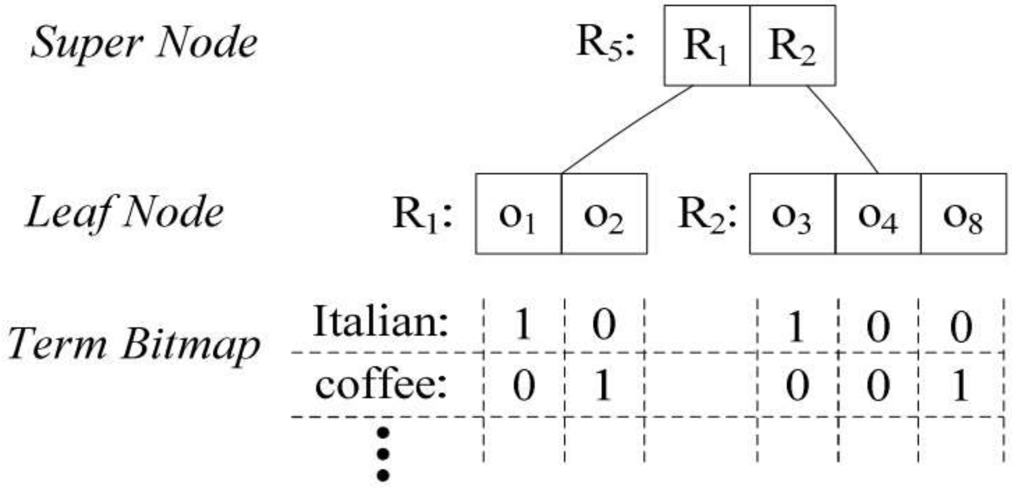
IR²-tree

- Signature + R-tree



SKI (Spatial-Keyword Indexing)

- **Bitmap + R-tree**



WIR-tree

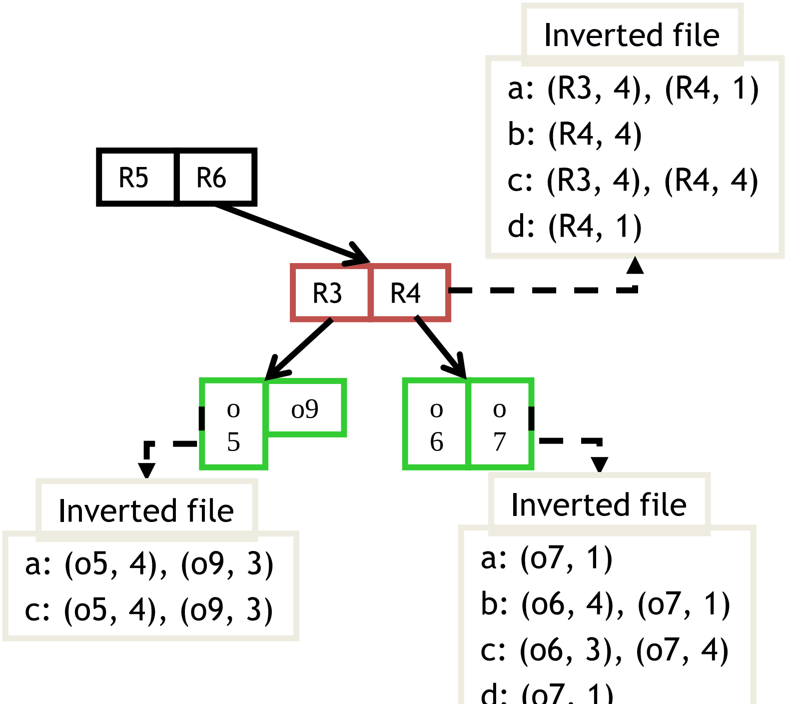
- **R-tree + inverted bitmaps**
  - Variant of IR-tree
- **Idea**
  - Consider the word frequency
  - Recursively partition objects by keyword frequency

IR-tree

- Augment each node of R-tree with a summary of the text content of the objects in the sub-tree

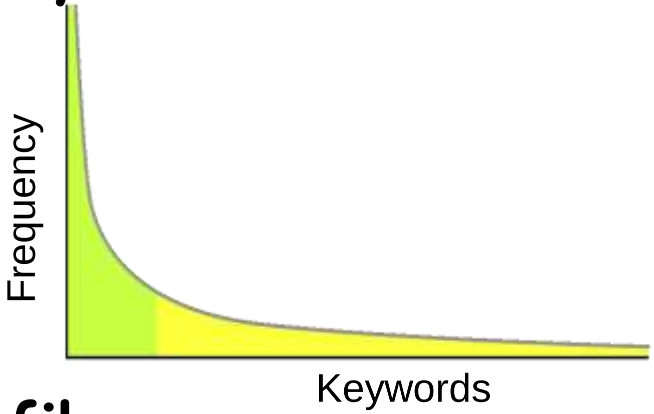
Object descriptions

	o5	o6	o7	o9
a	4	0	1	3
b	0	4	1	0
c	4	3	4	3
d	0	0	1	0



S2I (spatial inverted index)

- **Skewed distribution of keywords**



- **S2I: R-tree + inverted file**
  - Build inverted index first
  - Build term frequency-aware spatial index
    - **Frequent keywords:** aggregated R-trees (aR-trees)
    - **Less frequent keywords:** blocks

SKQs in Euclidean space

- **Standard SKQs**
  - Boolean range query (BRQ)
    - ST, TS
    - R*-Tree-IF, IF-R*-tree
    - KR*-Tree
    - SKIF
  - Boolean kNN query (BkQ)
    - IR2-tree
    - SKI
    - WIR-tree
  - Top-k query (TkQ)
    - IR-tree
- **Advanced SKQs**
  - m-CK query
  - Reverse query
  - Moving query
  - Group query
  - Direction-aware query
  - Region of interest query
  - Why-not query
  - Similarity join query
  - …

Indices for SKQ in Euclidean space

Index	Spatial index	Textual Index	Combination	BkQ	TkQ	BRQ
ST	Grid	IF	Spatial-first			√
TS	Grid	IF	Text-first			√
IF-R*-Tree	R*-Tree	IF	Text-first	Δ		√
R*-Tree-IF	R*-Tree	IF	Spatial-first		Δ	√
SF2I	SFC	IF	Spatial-first			√
KR*-Tree	R*-Tree	IF	Tightly combined	Δ		√
IR^2-Tree	R-Tree	Bitmap	Tightly combined	√		Δ
IR-Tree	R-Tree	IF	Tightly combined	Δ	√	Δ
SKIF	Grid	IF	Tightly combined			√
SKI	R-Tree	Bitmap	Spatial-first	√		
S2I	R-Tree	IF	Text-first	Δ	√	Δ
WIR-Tree	R-Tree	Inv. Bitmap	Tightly combined	√		Δ
SFC-QUAD	SFC	IF	Tightly combined			√

Summary

- Basic Concepts
- Ordered Indexing
- B+-tree & B-tree Indices
- Static & Dynamic Hashing
- Ordered Indexing vs. Hashing
- Index Definition in SQL
- Multiple-key Access

Assignments

- Practice exercises: 14.3, 14.4
- Exercises: 12.20

- Submission DDL: 12:49pm, May 15

End of Lecture 8