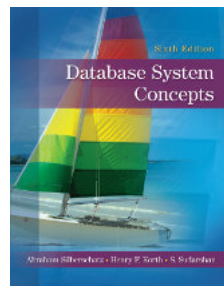


Database index structures

From: Database System
Concepts, 6th edition
Avi Silberschatz, Henry
Korth, S. Sudarshan
McGraw-Hill



Architectures for Massive DM
D&K / UPSay 2015-2016

Ioana Manolescu

1



Chapter 12: Indexing and Hashing

- Basic Concepts
- Ordered Indices
- B+-Tree Index Files
- B-Tree Index Files
- Static Hashing
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing



Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
 - E.g., author catalog in library
- **Search Key** - attribute to set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

search-key	pointer
------------	---------

- Index files are typically much smaller than the original file
- Two basic kinds of indices:
 - **Ordered indices:** search keys are stored in sorted order
 - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".



Index Evaluation Metrics

- Access types supported efficiently. E.g.,
 - records with a specified value in the attribute
 - or records with an attribute value falling in a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead



Ordered Indices

- In an **ordered index**, index entries are stored sorted on the search key value. E.g., author catalog in library.
- **Primary index**: in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
 - Also called **clustering index**
 - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called **non-clustering index**.
- **Index-sequential file**: ordered sequential file with a primary index.



Dense Index Files

- **Dense index** — Index record appears for every search-key value in the file.
- E.g. index on *ID* attribute of *instructor* relation

10101	→	10101	Srinivasan	Comp. Sci.	65000	→
12121	→	12121	Wu	Finance	90000	→
15151	→	15151	Mozart	Music	40000	→
22222	→	22222	Einstein	Physics	95000	→
32343	→	32343	El Said	History	60000	→
33456	→	33456	Gold	Physics	87000	→
45565	→	45565	Katz	Comp. Sci.	75000	→
58583	→	58583	Califieri	History	62000	→
76543	→	76543	Singh	Finance	80000	→
76766	→	76766	Crick	Biology	72000	→
83821	→	83821	Brandt	Comp. Sci.	92000	→
98345	→	98345	Kim	Elec. Eng.	80000	→



Dense Index Files (Cont.)

- Dense index on *dept_name*, with *instructor* file sorted on *dept_name*

Biology	76766	Crick	Biology	72000
Comp. Sci.	10101	Srinivasan	Comp. Sci.	65000
Elec. Eng.	45565	Katz	Comp. Sci.	75000
Finance	83821	Brandt	Comp. Sci.	92000
History	98345	Kim	Elec. Eng.	80000
Music	12121	Wu	Finance	90000
Physics	76543	Singh	Finance	80000
	32343	El Said	History	60000
	58583	Califieri	History	62000
	15151	Mozart	Music	40000
	22222	Einstein	Physics	95000
	33465	Gold	Physics	87000



Sparse Index Files

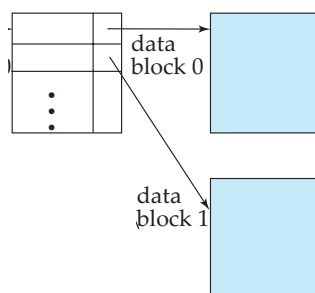
- **Sparse Index:** contains index records for only some search-key values.
 - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value K we:
 - Find index record with largest search-key value $< K$
 - Search file sequentially starting at the record to which the index record points

10101	10101	Srinivasan	Comp. Sci.	65000
32343	12121	Wu	Finance	90000
76766	15151	Mozart	Music	40000
	22222	Einstein	Physics	95000
	32343	El Said	History	60000
	33456	Gold	Physics	87000
	45565	Katz	Comp. Sci.	75000
	58583	Califieri	History	62000
	76543	Singh	Finance	80000
	76766	Crick	Biology	72000
	83821	Brandt	Comp. Sci.	92000
	98345	Kim	Elec. Eng.	80000

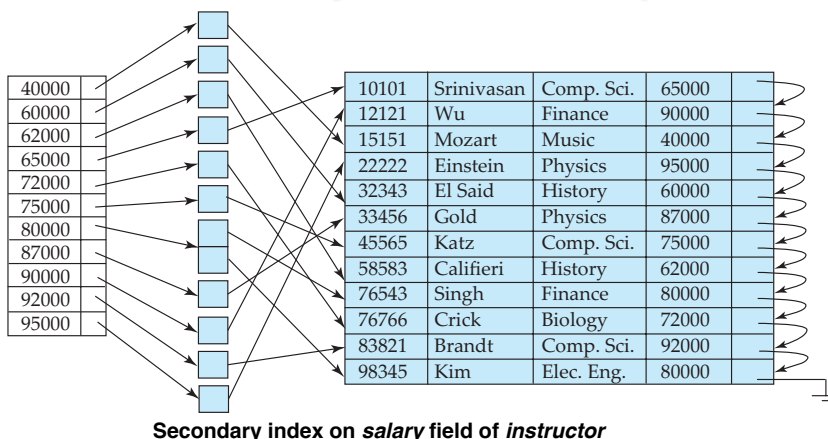


Sparse Index Files (Cont.)

- Compared to dense indices:
 - Less space and less maintenance overhead for insertions and deletions.
 - Generally slower than dense index for locating records.
- **Good tradeoff:** sparse index with an index entry for every block in file, corresponding to least search-key value in the block.



Secondary Indices Example



Secondary index on *salary* field of *instructor*

- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense



Primary and Secondary Indices

- Indices offer substantial benefits when searching for records.
- BUT: Updating indices imposes overhead on database modification: when a file is modified, every index on the file must be updated
- 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
 - Block fetch requires about 5 to 10 milliseconds, versus about 100 nanoseconds for memory access



Index Update: Deletion

- If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.

10101	→	10101	Srinivasan	Comp. Sci.	65000	
32343	→	12121	Wu	Finance	90000	
76766	→	15151	Mozart	Music	40000	
		22222	Einstein	Physics	95000	
		32343	El Said	History	60000	
		33456	Gold	Physics	87000	
		45565	Katz	Comp. Sci.	75000	
		58583	Califieri	History	62000	
		76543	Singh	Finance	80000	
		76766	Crick	Biology	72000	
		83821	Brandt	Comp. Sci.	92000	
		98345	Kim	Elec. Eng.	80000	

- **Dense indices** – deletion of search-key is similar to file record 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 (in search-key order).
 - If the next search-key value already has an index entry, the entry is deleted instead of being replaced.



Index Update: Insertion

■ Insertion:

- Perform a lookup using the search-key value appearing in the record to be inserted.
- **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.
 - ▶ If a new block is created, the first search-key value appearing in the new block is inserted into the index.



Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.
 - Example 1: In the *instructor* relation stored sequentially by ID, we may want to find all instructors in a **particular department**
 - Example 2: as above, but where we want to find all instructors with a **specified salary** or with **salary in a specified range of values**
- We can have a secondary index with an index record for each search-key value



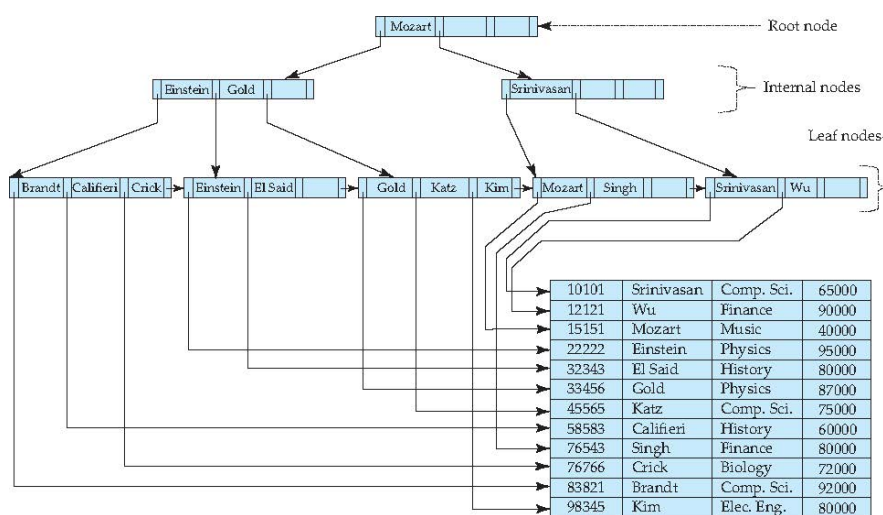
B+-Tree Index Files

B+-tree indices are an alternative to indexed-sequential files.

- Disadvantage of indexed-sequential files
 - performance degrades as file grows, since many overflow blocks get created.
 - Periodic reorganization of entire file is required.
- Advantage of B+-tree index files:
 - automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
 - Reorganization of entire file is not required to maintain performance.
- (Minor) disadvantage of B+-trees:
 - extra insertion and deletion overhead, space overhead.
- Advantages of B+-trees outweigh disadvantages
 - B+-trees are used extensively



Example of B+-Tree





B+-Tree Index Files (Cont.)

A B+-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
- A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values
- Special cases:
 - If the root is not a leaf, it has at least 2 children.
 - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n-1)$ values.



B+-Tree Node Structure

- Typical node

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

- K_i are the search-key values
- P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

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

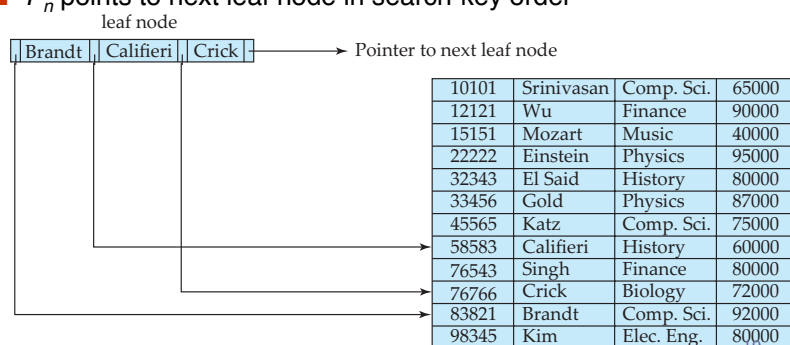
(Initially assume no duplicate keys, address duplicates later)



Leaf Nodes in B⁺-Trees

Properties of a leaf node:

- For $i = 1, 2, \dots, n-1$, pointer P_i points to a file record with search-key value K_i
- If L_i, L_j are leaf nodes and $i < j$, L_i 's search-key values are less than or equal to L_j 's search-key values
- P_n points to next leaf node in search-key order



Database System Concepts - 6th Edition

11.19

©Silberschatz, Korth and Sudarshan



Non-Leaf Nodes in B⁺-Trees

- For a non-leaf node with m 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 have values greater than or equal to K_{n-1}



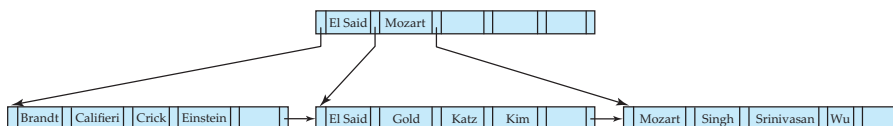
Database System Concepts - 6th Edition

11.20

20
©Silberschatz, Korth and Sudarshan



Example of B⁺-tree



B⁺-tree for *instructor* file ($n = 6$)

- Leaf nodes must have between 3 and 5 values ($\lceil (n-1)/2 \rceil$ and $n-1$, with $n = 6$).
- Non-leaf nodes other than root must have between 3 and 6 children ($\lceil n/2 \rceil$ and n with $n = 6$).
- Root must have at least 2 children.



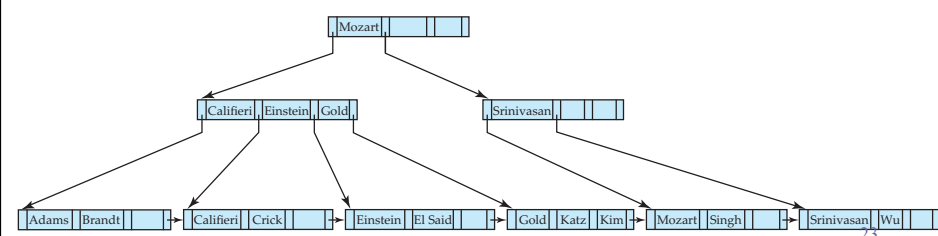
Observations about B⁺-trees

- Since the inter-node connections are done by pointers, “logically” close blocks need not be “physically” close.
- The B⁺-tree contains a **relatively small number of levels**
 - ▶ 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
 - ▶ .. etc.
- If there are K search-key values in the file, the tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
- Thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in **logarithmic time** (as we shall see).



Queries on B⁺-Trees

- Find record with search-key value V .
 1. $C = \text{root}$
 2. While C is not a leaf node {
 1. Let i be least value s.t. $V \leq K_i$.
 2. If no such exists, set $C = \text{last non-null pointer in } C$
 3. Else { if $(V = K_i)$ Set $C = P_{i+1}$ else set $C = P_i$ }
 3. Let i be least value s.t. $K_i = V$
 4. If there is such a value i , follow pointer P_i to the desired record.
 5. Else no record with search-key value k exists.

Database System Concepts - 6th Edition

11.23

©Silberschatz, Korth and Sudarshan



Queries on B⁺-Trees (Cont.)

- If there are K search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- A node is generally the same size as a disk block, typically 4 kilobytes
 - 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.
- Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup
 - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

Database System Concepts - 6th Edition

11.24

24
©Silberschatz, Korth and Sudarshan



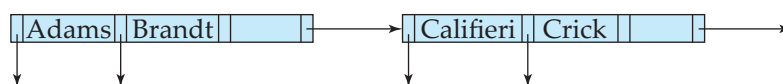
Updates on B+-Trees: Insertion

1. Find the leaf node in which the search-key value would appear
2. If the search-key value is already present in the leaf node **then**
 1. Add record to the file
 2. If necessary add a pointer to the bucket.
3. **Else**
 1. add the record to the main file (and create a bucket if necessary)
 2. If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
 3. Otherwise, split the node (along with the new (key-value, pointer) entry) as discussed in the next slide.

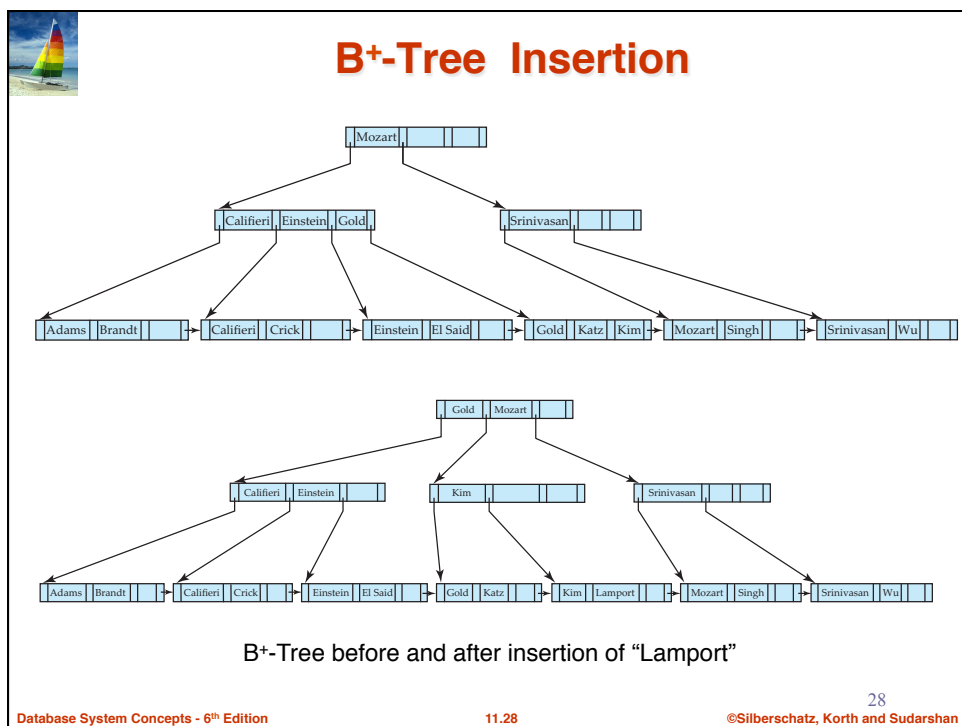
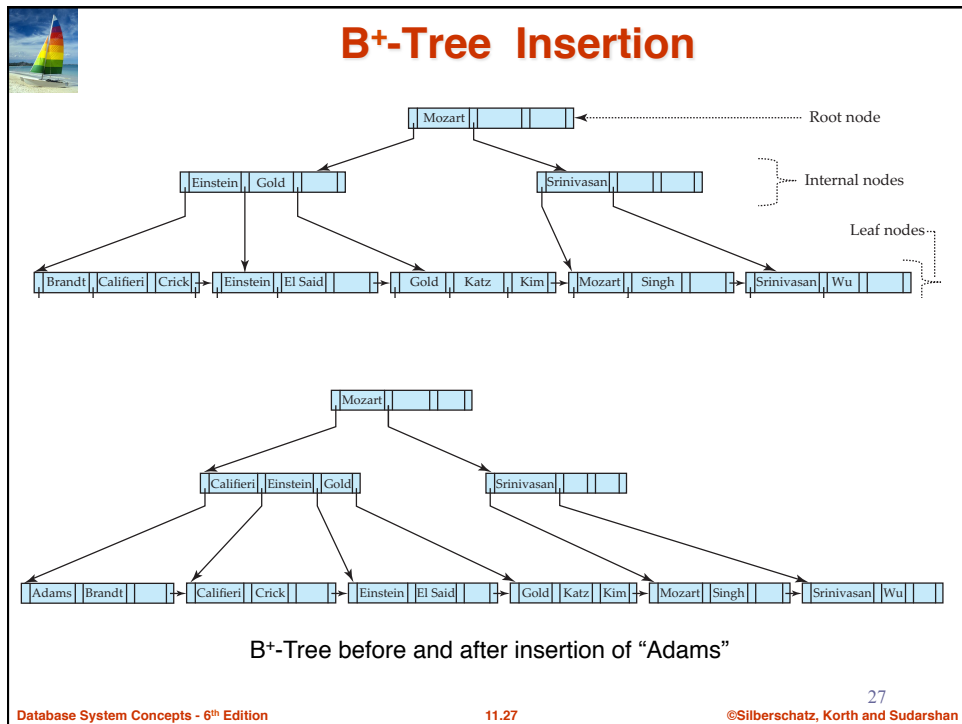


Updates on B+-Trees: Insertion (Cont.)

- 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 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 may be split increasing the height of the tree by 1.



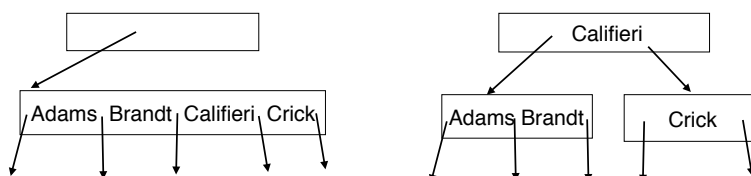
Result of splitting node containing Brandt, Califieri and Crick on inserting Adams
 Next step: insert entry with (Califieri, pointer-to-new-node) into parent





Insertion in B⁺-Trees (Cont.)

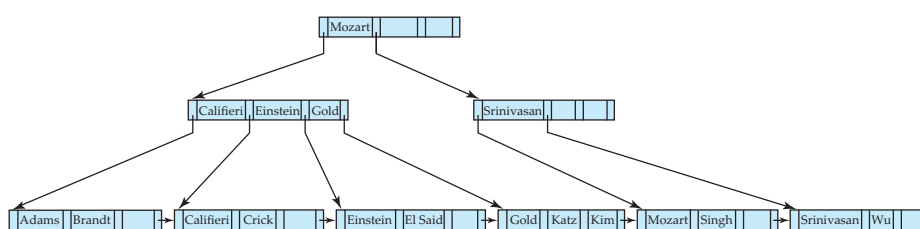
- Splitting a non-leaf node: when inserting (k,p) into an already full internal node N
 - Copy N to an in-memory area M with space for n+1 pointers and n keys
 - Insert (k,p) into M
 - Copy $P_1, K_1, \dots, K_{\lfloor n/2 \rfloor - 1}, P_{\lfloor n/2 \rfloor}$ from M back into node N
 - Copy $P_{\lfloor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

Database System Concepts - 6th Edition

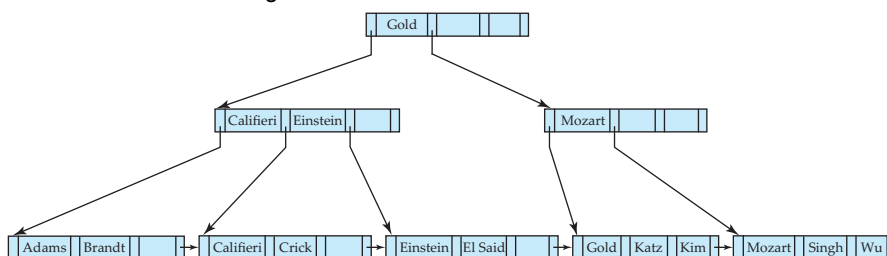
11.29

29
©Silberschatz, Korth and Sudarshan

Examples of B⁺-Tree Deletion



Before and after deleting "Srinivasan"



- Deleting "Srinivasan" causes merging of under-full leaves

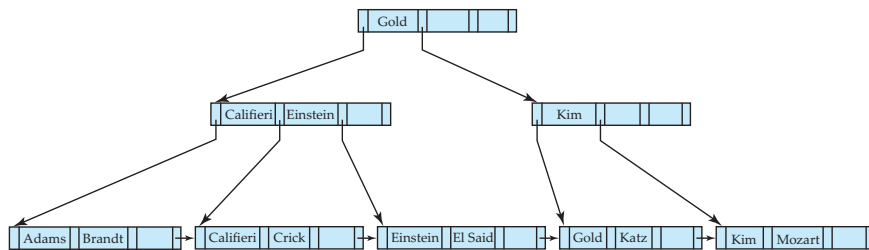
Database System Concepts - 6th Edition

11.30

30
©Silberschatz, Korth and Sudarshan



Examples of B+-Tree Deletion (Cont.)

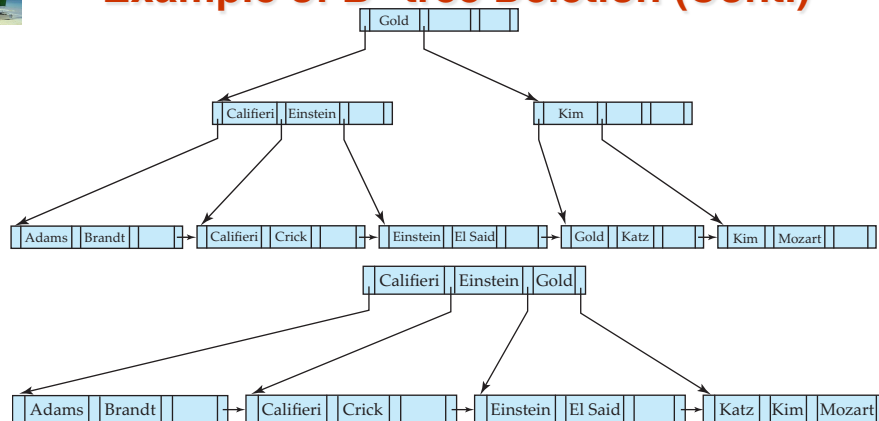


Deletion of “Singh” and “Wu” from result of previous example

- Leaf containing Singh and Wu became underfull, and borrowed a value Kim from its left sibling
- Search-key value in the parent changes as a result



Example of B+-tree Deletion (Cont.)



Before and after deletion of “Gold” from earlier example

- Node with Gold and Katz became underfull, and was merged with its sibling
- Parent node becomes underfull, and is merged with its sibling
 - Value separating two nodes (at the parent) is pulled down when merging
- Root node then has only one child, and is deleted



Updates on B⁺-Trees: Deletion

- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- 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.



Updates on B⁺-Trees: Deletion

- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not 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.

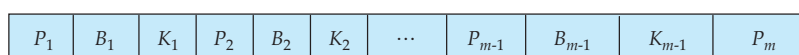


B-Tree Index Files

- Similar to B+-tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.
- Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.
- Generalized B-tree leaf node



(a)

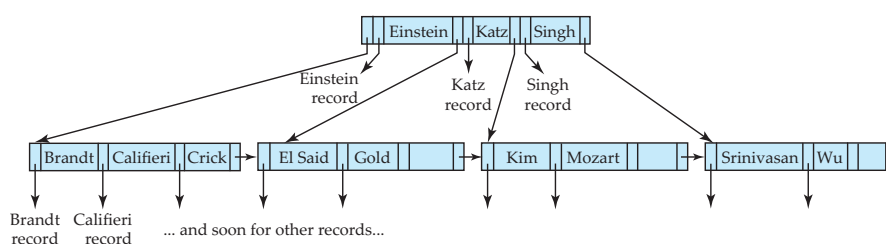


(b)

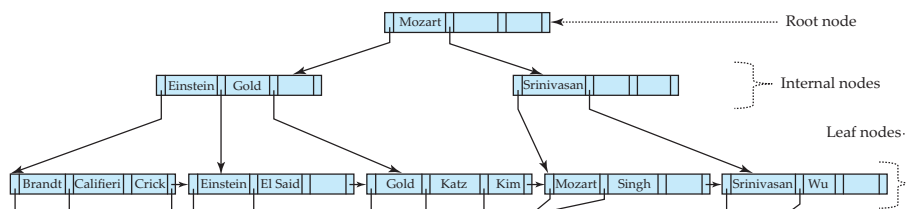
- Nonleaf node – pointers B_i are the bucket or file record pointers.



B-Tree Index File Example



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





B-Tree Index Files (Cont.)

- Advantages of B-Tree indices:
 - May use less tree nodes than a corresponding B⁺-Tree.
 - Sometimes possible to find search-key value before reaching leaf node.
- Disadvantages of B-Tree indices:
 - Only 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 corresponding B⁺-Tree
 - Insertion and deletion more complicated than in B⁺-Trees
 - Implementation is harder than B⁺-Trees.
- Typically, advantages of B-Trees do not outweigh disadvantages.



Multiple-Key Access

- Use multiple indices for certain types of queries.
- Example:


```
select ID
from instructor
where dept_name = "Finance" and salary = 80000
```
- Possible strategies for processing query using indices on single attributes:
 1. Use index on *dept_name* to find instructors with department name Finance; test *salary = 80000*
 2. Use index on *salary* to find instructors with a salary of \$80000; test *dept_name = "Finance"*.
 3. Use *dept_name* index to find pointers to all records pertaining to the "Finance" department. Similarly use index on *salary*. Take intersection of both sets of pointers obtained.



Indices on Multiple Keys

- **Composite search keys** are search keys containing more than one attribute
 - E.g. (*dept_name*, *salary*)
- Lexicographic ordering: $(a_1, a_2) < (b_1, b_2)$ if either
 - $a_1 < b_1$, or
 - $a_1 = b_1$ and $a_2 < b_2$



Indices on Multiple Attributes

Suppose we have an index on combined search-key
(*dept_name*, *salary*).

- With the **where** clause
where *dept_name* = "Finance" **and** *salary* = 80000
 the index on (*dept_name*, *salary*) can be used to fetch only records that satisfy both conditions.
 - Using separate indices is less efficient — we may fetch many records (or pointers) that satisfy only one of the conditions.
- Can also efficiently handle
where *dept_name* = "Finance" **and** *salary* < 80000
 - May fetch many records that satisfy the first but not the second condition
- But cannot efficiently handle
where *dept_name* < "Finance" **and** *balance* = 80000
 - May fetch many records that satisfy the first but not the second condition



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 entire bucket has to be searched sequentially to locate a record.



Example of Hash File Organization

Hash file organization of *instructor* file, using *dept_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. $h(\text{Music}) = 1$ $h(\text{History}) = 2$
 $h(\text{Physics}) = 3$ $h(\text{Elec. Eng.}) = 3$



Example of Hash File Organization

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

Hash file organization of *instructor* file, using *dept_name* as key (see previous slide for details).



Hash Functions

- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- 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.
 - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned. .



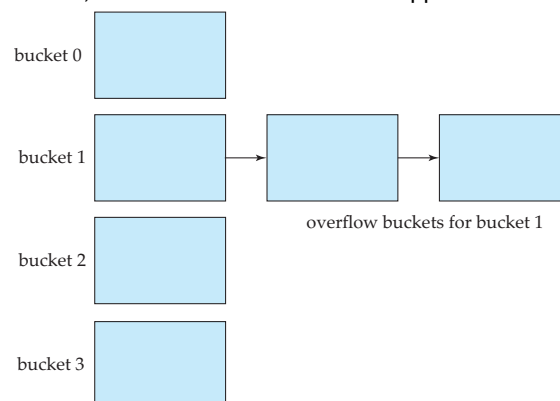
Handling of Bucket Overflows

- 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.
- Above scheme is called **closed hashing**.
 - An alternative, called **open hashing**, which does not use overflow buckets, is not suitable for database applications.



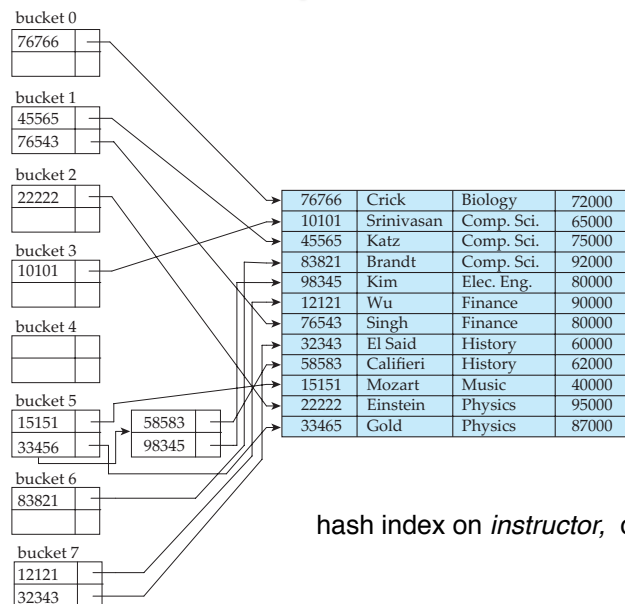


Hash Indices

- 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





Deficiencies of Static Hashing

- In static hashing, function h maps search-key values to a fixed set of B of bucket addresses. Databases grow or shrink with time.
 - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
 - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
 - If database shrinks, again space will be wasted.
- One solution: periodic re-organization of the file with a new hash function
 - Expensive, disrupts normal operations
- Better solution: allow the number of buckets to be modified dynamically.

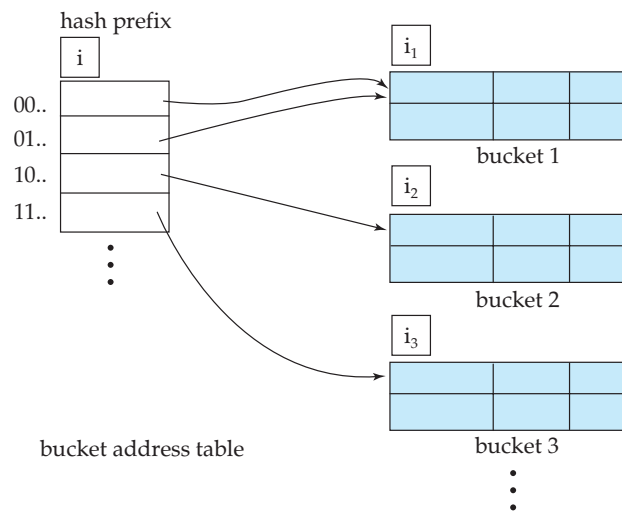


Dynamic Hashing

- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically
- **Extendable hashing** – one form of dynamic hashing
 - Hash function generates values over a large range — typically b -bit integers, with $b = 32$.
 - Buckets are created **on demand**
 - At any time **use only a prefix of the hash function result** 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 (why?)
 - 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



In this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$ (see next slide for details)



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)



Insertion in Extendable Hash Structure (Cont)

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 $i_j = i_z = (i_j + 1)$
 - Update the second half of the bucket address table entries originally pointing to j , to point to z
 - remove each record in bucket j and reinsert (in j or z)
 - 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)
 - If i reaches some limit b , or too many splits have happened in this insertion, create an overflow bucket
 - Else
 - 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.



Deletion in Extendable Hash Structure

- 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 having same value of i_j and same $i_j - 1$ prefix, 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



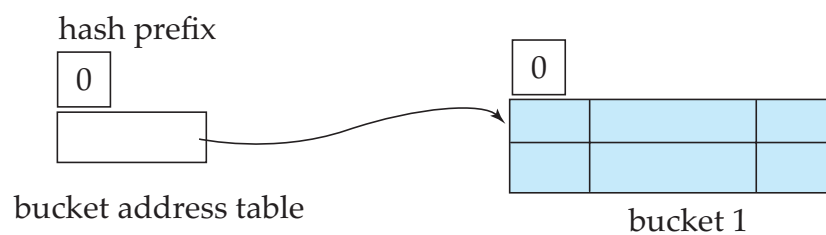
Use of Extendable Hash Structure: Example

<i>dept_name</i>	<i>h(dept_name)</i>
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001



Example (Cont.)

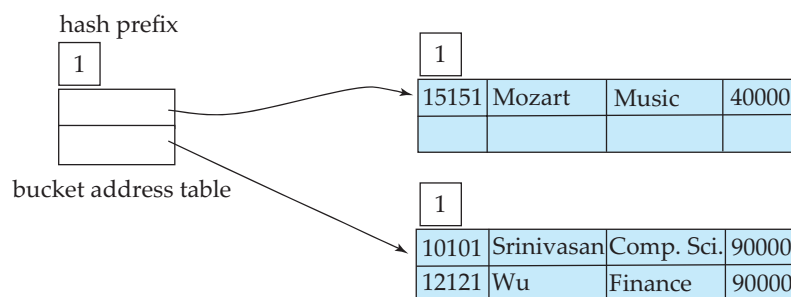
- Initial Hash structure; **bucket size = 2**





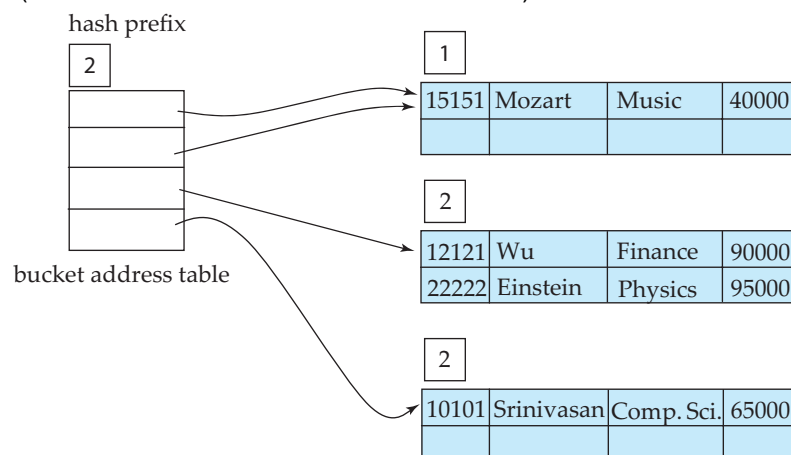
Example (Cont.)

- Hash structure after insertion of "Mozart", "Srinivasan", and "Wu" records



Example (Cont.)

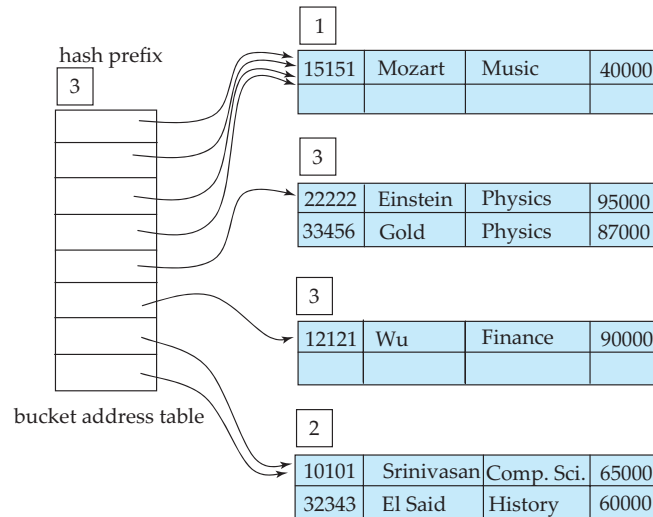
- Hash structure after insertion of Einstein record (assume it hashed with Wu and Srinivasan)





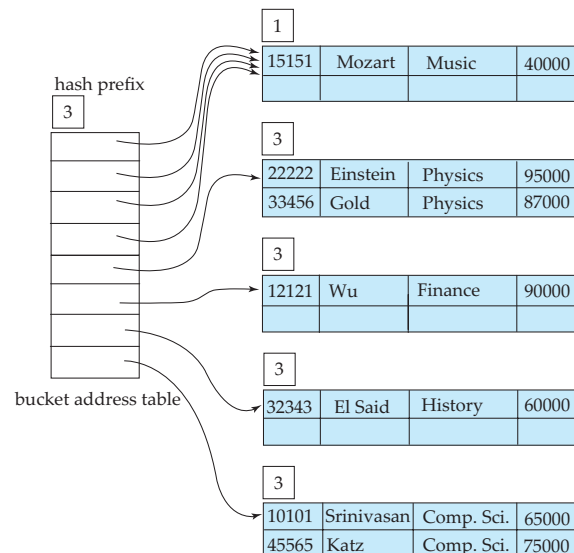
Example (Cont.)

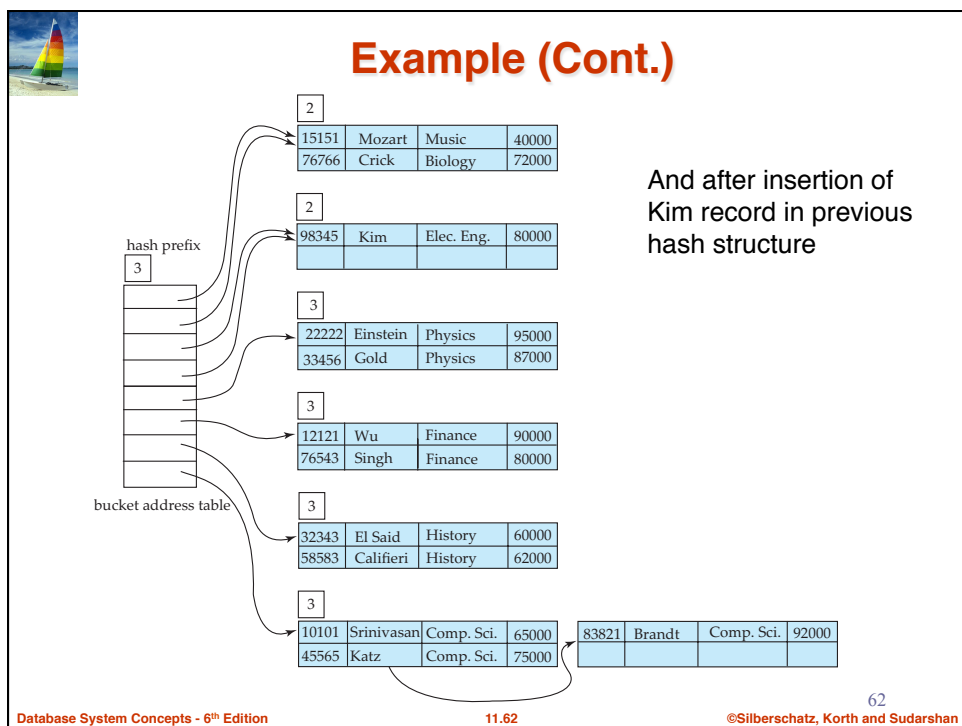
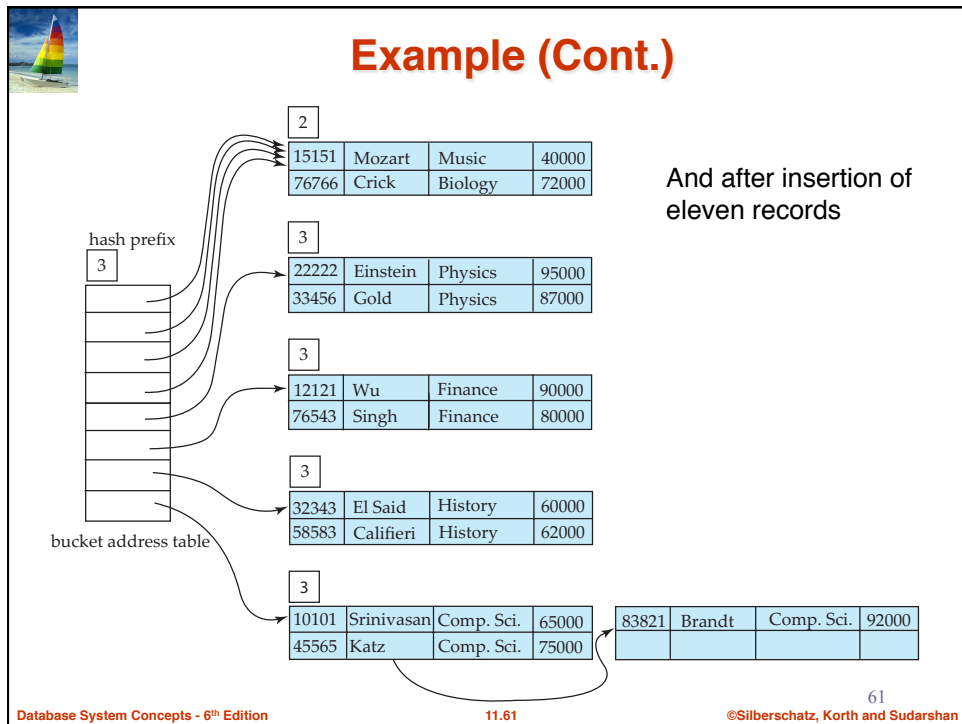
■ Hash structure after insertion of Gold and El Said records



Example (Cont.)

■ Hash structure after insertion of Katz record







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)
 - ▶ Cannot allocate very large contiguous areas on disk either
 - ▶ Solution: B⁺-tree structure to locate desired record in bucket address table
 - Changing size of bucket address table is an expensive operation
- Linear hashing is an alternative mechanism
 - Allows incremental growth of its directory (equivalent to bucket address table)
 - At the cost of more bucket overflows



Comparison of Ordered Indexing and Hashing

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize 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 to be preferred
- In practice:
 - PostgreSQL supports hash indices, but discourages use due to poor performance
 - Oracle supports static hash organization, but not hash indices
 - SQLServer supports only B⁺-trees



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, say, 0
 - 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 = 4 levels)
- 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	ID	gender	income_level	Bitmaps for gender	Bitmaps for income_level
		m		10010	
		f		01101	
0	76766	m	L1		L1 10100
1	22222	f	L2		L2 01000
2	12121	f	L1		L3 00001
3	15151	m	L4		L4 00010
4	58583	f	L3		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. $100110 \text{ AND } 110011 = 100010$
 $100110 \text{ OR } 110011 = 110111$
 $\text{NOT } 100110 = 011001$
 - Males with income level L1: $10010 \text{ AND } 10100 = 10000$
 - ▶ 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
 - ▶ $\text{not}(A=v): (\text{NOT } \text{bitmap-}A-v) \text{ AND ExistenceBitmap}$
- Should keep bitmaps for all values, even null value
 - To correctly handle SQL null semantics for $\text{NOT}(A=v)$:
 - ▶ intersect above result with $(\text{NOT } \text{bitmap-}A\text{-Null})$

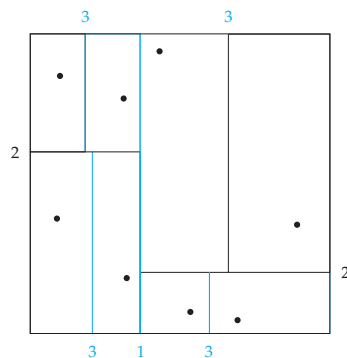


Chapter 25: Indexing of Spatial Data

- **k-d tree** - early structure used for indexing in multiple dimensions.
- Each level of a *k-d* tree partitions the space into two.
 - choose one dimension for partitioning at the root level of the tree.
 - choose another dimensions for partitioning in nodes at the next level and so on
 - **cycling through the dimensions.**
- In each node, approximately half of the points stored in the sub-tree fall on one side and half on the other.
- Partitioning stops when a node has less than a given maximum number of points.
- The **k-d-B tree** extends the *k-d* tree to allow multiple child nodes for each internal node; well-suited for secondary storage.



Division of Space by a k-d Tree



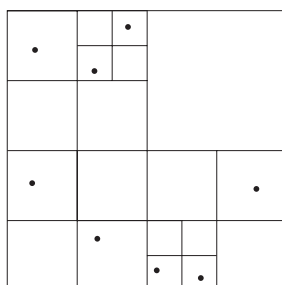
- Each line in the figure (other than the outside box) corresponds to a node in the *k-d* tree.
 - The maximum number of points in a leaf node has been set to 1.
- The numbering of the lines in the figure indicates the level of the tree at which the corresponding node appears.



Division of Space by Quadrees

Quadrees

- Each node of a quadtree is associated with a rectangular region of space; the top node is associated with the entire target space.
- Each non-leaf node divides its region into four equal sized quadrants
 - Correspondingly each such node has four child nodes corresponding to the four quadrants and so on
- Leaf nodes have between zero and some fixed maximum number of points (set to 1 in example).



Quadrees (Cont.)

- **PR quadtree**: stores points; space is divided based on regions, rather than on the actual set of points stored.
- **Region quadrees** store array (raster) information.
 - A node is a leaf node if all the array values in the region that it covers are the same. Otherwise, it is subdivided further into four children of equal area, and is therefore an internal node.
 - Each node corresponds to a sub-array of values.
 - The sub-arrays corresponding to leaves either contain just a single array element, or have multiple array elements, all of which have the same value.
- Extensions of *k-d* trees and PR quadrees have been proposed to index line segments and polygons
 - Require splitting segments/polygons into pieces at partitioning boundaries
 - Same segment/polygon may be represented at several leaf nodes




R-Trees

- **R-trees** are a N-dimensional extension of B⁺-trees, useful for indexing sets of rectangles and other polygons.
- Supported in many modern database systems, along with variants like R⁺-trees and R*-trees.
- Basic idea: generalize the notion of a one-dimensional interval associated with each B⁺-tree node to an N-dimensional interval, that is, an N-dimensional rectangle.
- Will consider only the two-dimensional case ($N = 2$)
 - generalization for $N > 2$ is straightforward, although R-trees work well only for relatively small N



R Trees (Cont.)

- A rectangular **bounding box** is associated with each tree node.
 - Bounding box of a leaf node is a minimum sized rectangle that contains all the rectangles/polygons associated with the leaf node.
 - The bounding box associated with a non-leaf node contains the bounding box associated with all its children.
 - Bounding box of a node serves as its key in its parent node (if any)
 - *Bounding boxes of children of a node are allowed to overlap*
- A polygon is stored only in one node, and the bounding box of the node must contain the polygon.
 - The storage efficiency of R-trees is better than that of k-d trees or quadrees since a polygon is stored only once.



Example R-Tree

■ A set of rectangles (solid line) and the bounding boxes (dashed line) of the nodes of an R-tree for the rectangles. The R-tree is shown on the right.

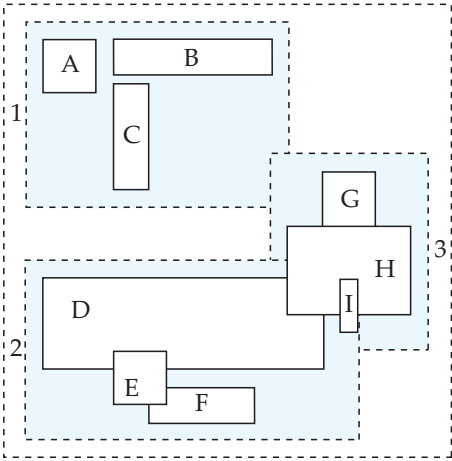
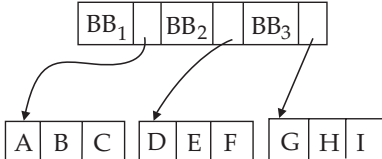



Diagram showing rectangles A through I and their bounding boxes 1, 2, and 3. Bounding box 1 contains A, B, and C. Bounding box 2 contains D, E, and F. Bounding box 3 contains G, H, and I.



R-tree structure diagram. Root node has three children: BB₁, BB₂, and BB₃. BB₁ points to leaf node [A B C], BB₂ points to leaf node [D E F], and BB₃ points to leaf node [G H I].

Database System Concepts - 6th Edition 11.75 75
©Silberschatz, Korth and Sudarshan



Search in R-Trees

- To find data items (rectangles/polygons) intersecting (overlaps) a given query point/region, do the following, starting from the root node:
 - If the node is a leaf node, output the data items whose keys intersect the given query point/region.
 - Else, for each child of the current node whose bounding box overlaps the query point/region, recursively search the child
- Can be very inefficient in worst case since multiple paths may need to be searched
 - but works acceptably in practice.
- Simple extensions of search procedure to handle predicates *contained-in* and *contains*

Database System Concepts - 6th Edition 11.76 76
©Silberschatz, Korth and Sudarshan



Insertion in R-Trees

- To insert a data item:
 - Find a leaf to store it, and add it to the leaf
 - ▶ To find leaf, follow a child (if any) whose bounding box contains bounding box of data item, else child whose overlap with data item bounding box is maximum
 - Handle overflows by splits (as in B+-trees)
 - ▶ Split procedure is different though (see below)
 - Adjust bounding boxes starting from the leaf upwards
- Split procedure:
 - Goal: divide entries of an overfull node into two sets such that the bounding boxes have minimum total area
 - ▶ This is a heuristic. Alternatives like minimum overlap are possible
 - Finding the “best” split is expensive; several heuristics



Deleting in R-Trees

- Deletion of an entry in an R-tree done much like a B+-tree deletion.
 - In case of underfull node, borrow entries from a sibling if possible, else merging sibling nodes
 - Alternative approach removes all entries from the underfull node, deletes the node, then reinserts all entries