Chapter 11: 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
- Index Definition in SQL
- Multiple-Key Access

Basic Concepts

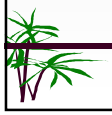
- Indexing mechanisms used to speed up access to desired data
  - E.G., Author catalog in library
- Search key - attribute to set of attributes used to look up records in a file
- An index file consists of records (called index entries) of the form

search-key	pointer

- Index files are typically much smaller than the original file

Two basic kinds of indices:

1. Ordered indices: search keys are stored in sorted order
2. Hash indices: search keys are distributed uniformly across "buckets" using a "hash function"



Index Evaluation Metrics

- Access (retrieve) records
  - with a specified value in the attribute or
  - within a specified range of values.
- Insertion time
- Deletion time
- Update time (delete & insert)
- Space overhead (10-15%)

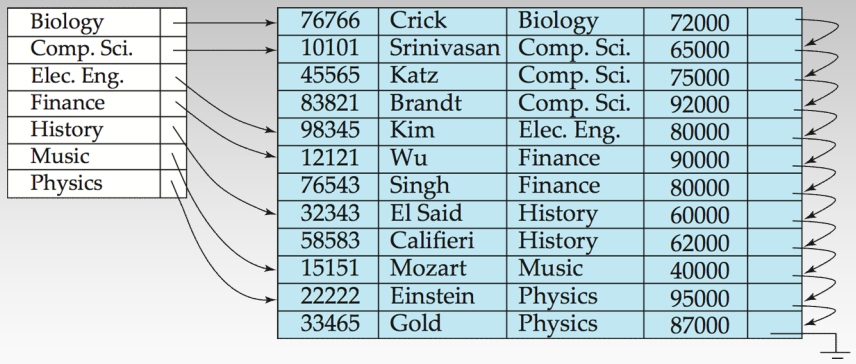
retrieve is the most frequent because it is also used for delete, update, and perhaps for insert

1. Ordered Indices

- Index entries are stored sorted on the search key value
  - E.g., author catalog in library
- Clustering index: a sequentially ordered file is searched using the sequential order of the file.
  - Also called primary index
- Secondary index: search keys in the file are not ordered
  - Also called non-clustering index
- Index-sequential file: ordered sequential file with a primary index.

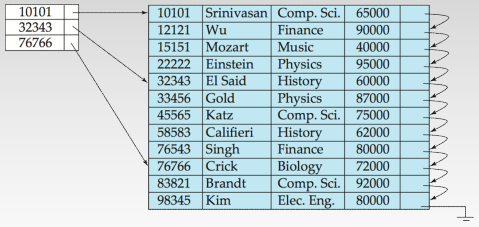
Dense Index Files

- **Dense index — all keys are in the index file.**



Sparse Index Files

- **Sparse Index: only some keys are in the index file**
  - **Applicable when records are sequentially ordered on search-key**
- **To locate a record with 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**

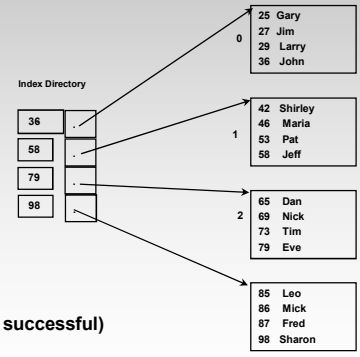


- **Less space than a dense index and less maintenance overhead for insertions and deletions**
- **Generally slower than a dense index for locating records.**

Example of a Good Sparse Index Files

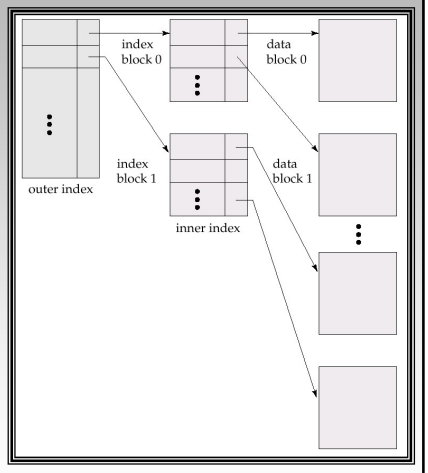
Index Sequential Access Method (ISAM)

- file is maintained sorted (inherits disadvantages of sequential file)
- records are divided into
  - blocks
  - an index table with the highest key value is created
- search the index table for the first key that covers the search key and go to the corresponding block
- most common examples
  - phone book, dictionary
- advantages:
  - sequential access in the order of the key
  - fast index search
- disadvantages
  - deletions fragment the space
  - insertion require shifting
  - all these updates may affect the index directory
- search (b blocks in the index directory
  - sequential: b/2 (+1 for the block with the record if successful)
  - binary: logb +1 -"-



Multilevel Index

- If primary index does not fit in memory, access becomes expensive
- To reduce number of disk accesses to index records, store the primary index on disk as a sequential file and construct a sparse index on it.
  - Outer index – an ISAM index of primary index
  - Inner index – the primary index file
- If even outer index is too large to fit in main memory, add yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.

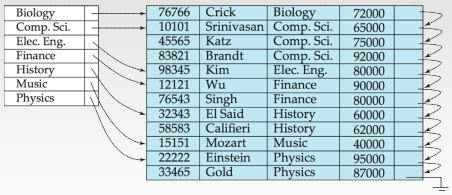


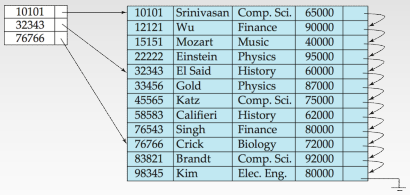
Review of Indices

- Ordered (sequential file) vs. unordered (hashed file)
- Sequential order of the file
  - (primary) key -- single entry in the index
  - Secondary index on a non-key attribute

76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
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

- Dense vs Sparse index

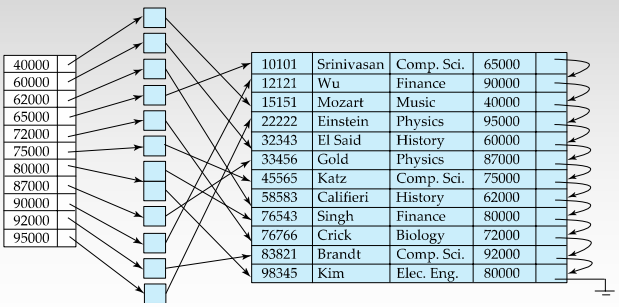




- Clustering vs Unclustering index

Secondary Indices

- find all the records on attributes other than the primary key
- Search returns 0 or more records
  - Single value or range search
  - index entry points to a bucket that contains pointers to all file records that satisfy the search value or range of values



Primary and Secondary Indices

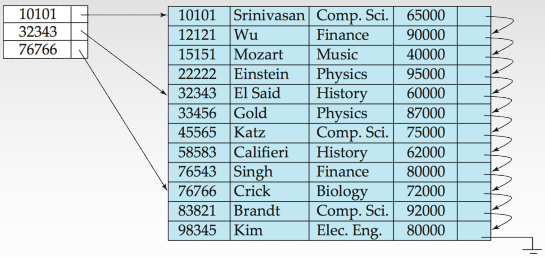
- Secondary indices have to be dense
- Indices offer substantial benefits when searching for records
- When a file is modified, every index on the file must be updated,
- Updating indices imposes overhead on database updates
- Ordered scan using primary index is efficient, but a ordered scan using a secondary index is expensive
  - each record access may fetch a new block from disk

Problems with Sequential Files

- Retrieve: search until the key value or a larger key value is found
  - Individual key access BAD
  - Scan the file in order of the key GOOD
- Insert: hard, all higher key records have to be shifted to place the new one
- Delete: leaves holes (if a delete bit is used) and leads to fragmentation
- Update: equivalent to delete & insert
  - update a non-key field may need shifting (as in insert) or fragmentation (as in delete)
- differential files: hold the recent updates until you reorganize off-line
- successful search: worst case all blocks, best case 1, average half
- unsuccessful search: -"-
- binary search: logN

Problems with Sequential Files-Deletion

- If deleted record was the only record in the file with its particular value, the value has to be deleted from the index also
- Single-level index deletion:
  - Dense indices – deletion of a value is similar to the file record deletion
  - Sparse indices – if the record value exists in the index, it is deleted and replaced with the next value from the file



Problems with Sequential Files-Insertion

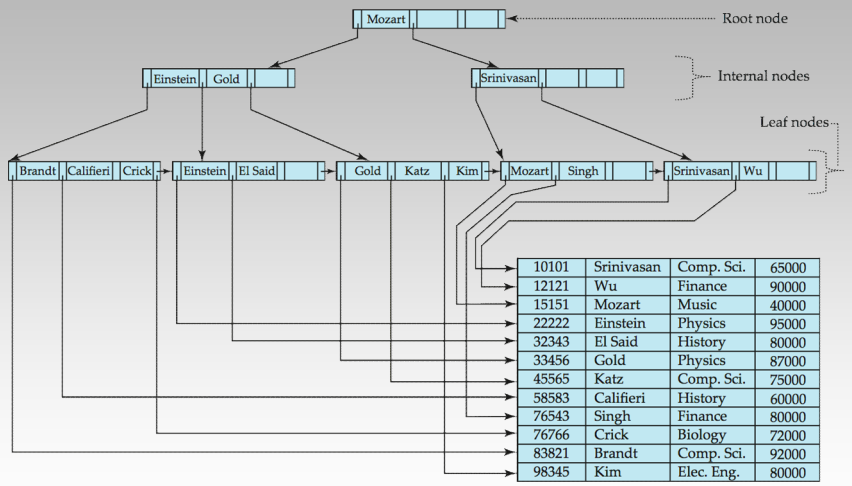
- Single-level index insertion:
  - Search for the value of the record to be inserted
  - Dense indices – if the key value is not found in the index, insert it.
  - Sparse indices
    - if the inserted record value falls within two existing index entry values (no split is needed) it is just inserted in the file and do nothing to the index
    - If a new bucket/block needs to be created, the first (last) key value appearing in the new block is inserted into the index.
- Multilevel insertion and deletion: algorithms are simple extensions of the single-level algorithms

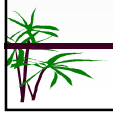
B+-Tree Index Files

B+-tree indices:
- an alternative to indexed-sequential files
- used for primary and secondary indexing

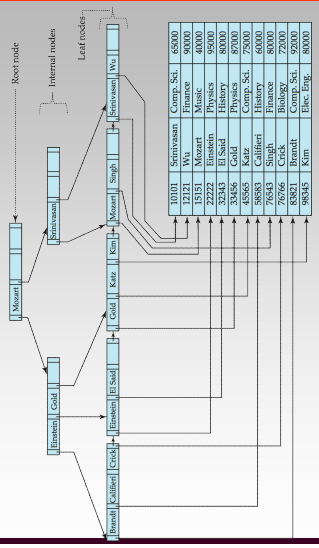
- Disadvantage of ISAM files
  - performance degrades as file grows, lots of overflow blocks
  - periodic reorganization of entire file is required
- Advantage of B+-tree index files:
  - automatically reorganizes itself with small, local, changes, during of insertions and deletions.
  - reorganization of entire file is not required to maintain performance.
- (Minor) disadvantage of B+-trees:
  - extra insertion and deletion overhead
- Advantages of B+-trees outweigh disadvantages
  - range search because of the key order
  - B+-trees are used extensively

Example of B+-Tree





Another View of a B+Tree



B+-Tree Index File

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

- **all paths from root to leaf are of the same length**
- **search (successful/unsuccessful) is uniform and logarithmic:**
  - **Equals the number of levels of the tree (1 for the root + height)**
- **the root has anywhere between 1 and n-1 keys**
- **all other nodes have anywhere between n/2 and n-1 keys (>= 50% space utilization)**

- **we compute the order n so that each node retrieval corresponds to one block I/O and contains as many keys as possible**
  - **parameters: B: size of a block in bytes (e.g. 8192)**
    K: size of the key
    P: size of a pointer (e.g. 4 Bytes)
  - **internal node: (n-1)*K + n*P <= B ==> n <= ⌈(B+K)/(K+P)⌉**

- **Examples: B=16384 K=8 P=4 n<= ⌈(16384+8)/(8+4)⌉ =16392/12 = 1366**
  B=16384 K=8 P=8 n<= ⌈(16384+8)/(8+8)⌉ =16392/16 = 1024

B$^+$-Tree Node Structure

- Typical node

P_1	K_1	P_2	...	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

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

  (Initially assume no duplicate keys, address duplicates later)

255

Non-Leaf Nodes in B$^+$-Trees

- Non leaf nodes form a multi-level sparse index on the leaf nodes. 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 $2 \le i \le n - 1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_i
  - All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}

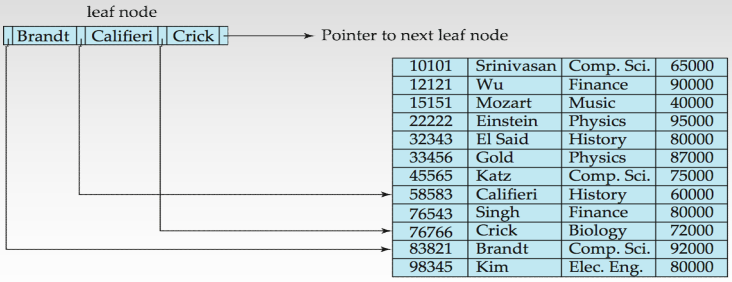
P_1	K_1	P_2	...	P_{n-1}	K_{n-1}	P_n

256

Leaf Nodes in B+-Trees

Properties of a leaf node:

- For $i = 1, 2, \ldots, n-1$, 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

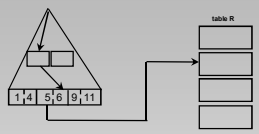


10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000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

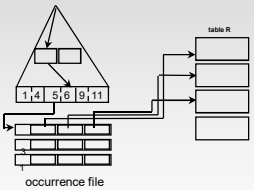
257

Leaf Nodes in B+-Trees

- a primary key index
  - single key values (ss#)



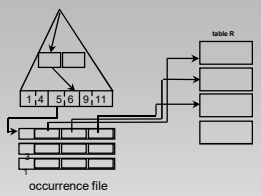
- secondary index (non-key)
  - multiple key values



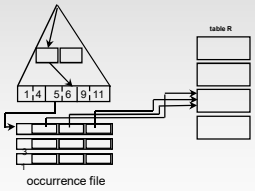
258

Clustering vs Non-clustering Index

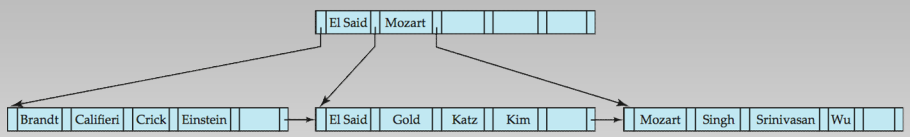
- Non-Clustering index
  - File is not sorted on the indexed attribute



- Clustering index
  - File is sorted on the indexed attribute



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

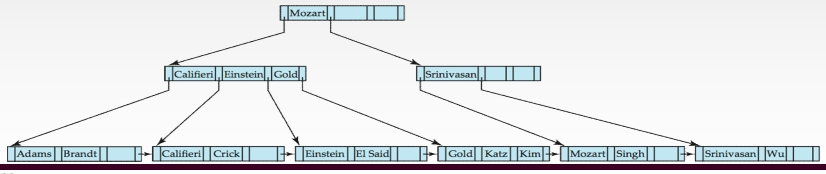
- Since the inter-node connections are done by pointers, "logically" close blocks need not be "physically" close
  - NO SHIFTING of blocks
- The non-leaf levels of the B$^+$-tree form a hierarchy of sparse indices
- The B$^+$-tree contains a relatively small number of levels
  - If there are K records in the file, the height is $\leq \lceil \log_{\lceil n/2 \rceil}(K) \rceil$

  thus search is efficient (logarithmic)
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time

Queries on B$^+$-Trees

- Find record with search-key value V.
  1. *C=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 = *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.

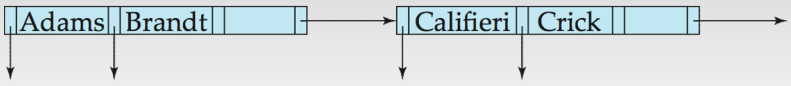


Updates on B$^+$-Trees: Insertion

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
   1. add record to the file
   2. add a pointer to the bucket.
3. If the search-key value is not present, then
   1. add the record to the 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.

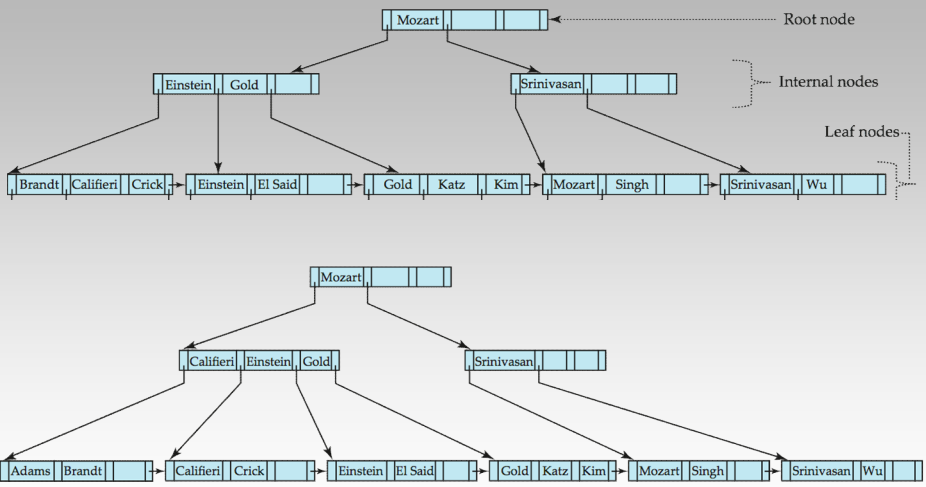
Updates on B$^+$-Trees: Insertion (Cont.)

- 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

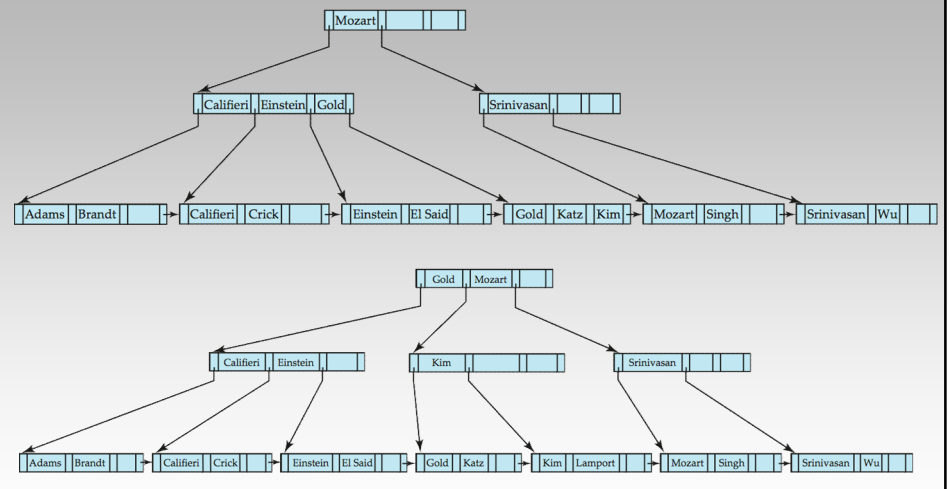
B+-Tree Insertion



B+-Tree before and after insertion of "Adams"

265

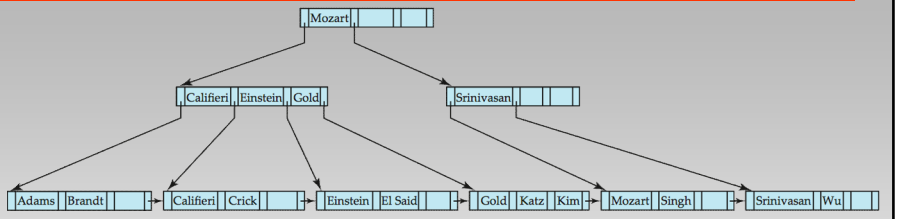
B+-Tree Insertion



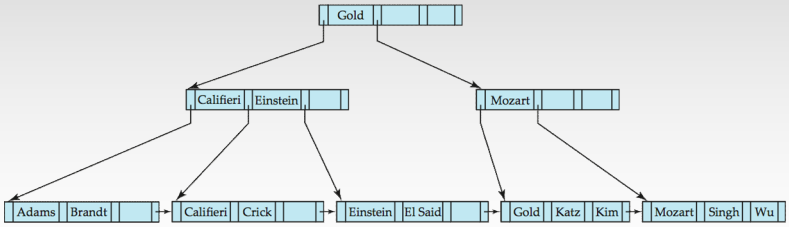
B+-Tree before and after insertion of "Lamport"

266

Examples of B+-Tree Deletion



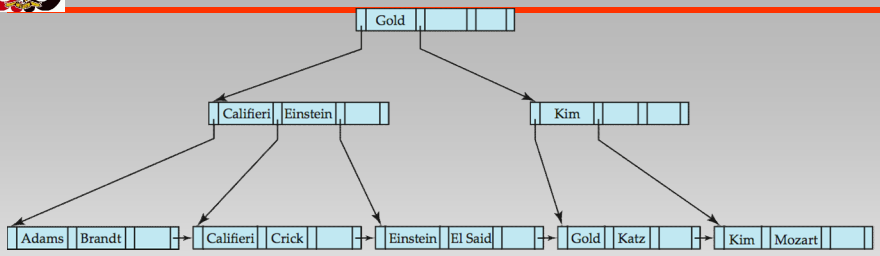
Before and after deleting "Srinivasan"



Deleting "Srinivasan" causes merging of underflow leaves

267

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**

268

Bulk Loading and Bottom-Up Build

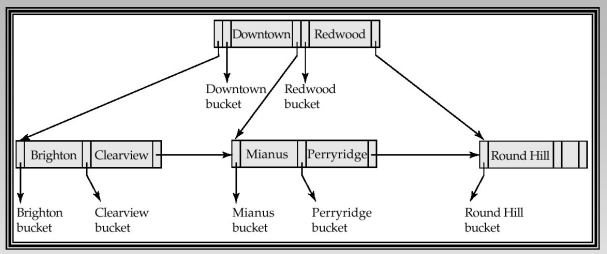
- Inserting entries one-at-a-time into a B^+-tree requires ≥ 1 IO per entry
  - can be very inefficient for loading a large number of entries at a time (bulk loading)
- Efficient alternative 1:
  - sort entries first
  - insert in sorted order
    - insertion will go to existing page (or cause a split)
    - much improved IO performance, but most leaf nodes half full
- Efficient alternative 2: Bottom-up B^+-tree construction
  - sort entries first
  - create tree layer-by-layer, starting with leaf level but fill them at a 80-85%
  - Implemented as part of bulk-load utility by most database systems

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.

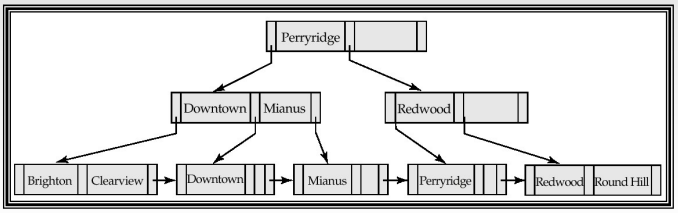
B-Tree Index File Example

B-tree



and

B+-tree on same data



B-Tree Index Files

- **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.**

2. Unordered Indexes: Hashed File

- divide the set of blocks into ***buckets***
- devise a hashing function that maps each key value into a bucket
  - V: set of key values
  - B: number of buckets
  - H: V → {0,1,2,…,B-1} Hashing function

Example: V: 9 digit SID
B: 1000
H: K MOD 1001

- search for, insert, delete, modify a key K do
  - H(K) to get the bucket number
  - search sequentially in the bucket (no structure within each bucket)
- selection of H: almost any function that generates "random" numbers in [0,B-1]
  - try to distribute evenly the keys into the B buckets
  - rule of thumb for MOD: prime number
- collisions: two or more key values go to the same bucket
  - too many collisions increases the search time degrades performance
  - no collisions means that each bucket has only (one or a few) key(s)

Hash Functions

- Worst hash function maps all search-key values to the same bucket;
  - H(K) = 3
  - Results in sequential search
- An ideal hash function is
  - Uniform: each bucket is assigned the same number of keys
  - Random: each bucket has the same number of records irrespective of the *actual* key value distribution
- Typical hash functions perform computation on the internal bits of the key

Example for a string search-key

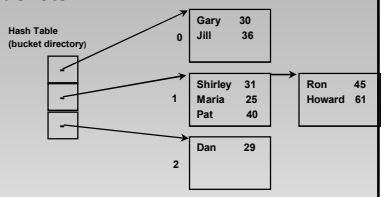
H(K)= SUM(the binary representations of all the characters in the string) MOD 1001

Hashed File

- hash table: holds the physical address of the buckets

Example:

EMP(ename,sal) H(sal): sal MOD 3



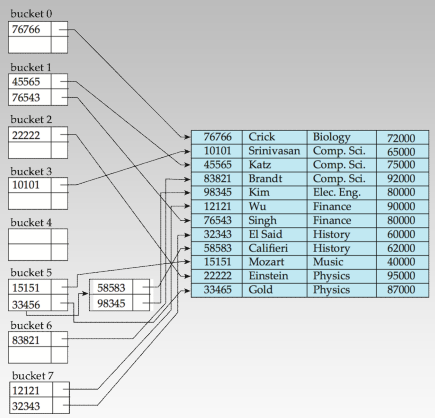
- overflow may occur for the following reasons:
  - too many records
  - poor hashing function
  - skewed data (too many values hash to the same bucket)
- overflow is handled by one of the two methods:
  - ***chaining*** of multiple blocks in a bucket
  - ***open addressing***: if the hashed bucket H(K) is full, put it in H(K)+1. If also full, in H(K)+2, etc. NOT USEFULE FOR DATABASES
  - ***double hashing***: hash once to H(K). If full, hash again with another H'(K). If still full, then apply any of the above methods
- performance depends on the ***loading factor***=# of records/(B*f) where f is the number of keys in a block
  - rule of thumb: when loading factor too high, double B and rehash

Search Cost of a Hashed File

- assume the hash table is in main memory
  - successful search:
    - Best case: 1 block
    - Worst case: all chained bucket blocks
    - average -" : half of worst case
  - unsuccessful search:
    - best, worst, average case: all chained bucket blocks
  - for loading factor of about 90% and a good hashing function the average access cost is 1.2 I/Os (blocks)
- Advantage of hashing: very fast for ***exact*** queries
- Disadvantage of hashing: because records are not sorted, we cannot do ***range*** queries

Hash Indices

- Another example:



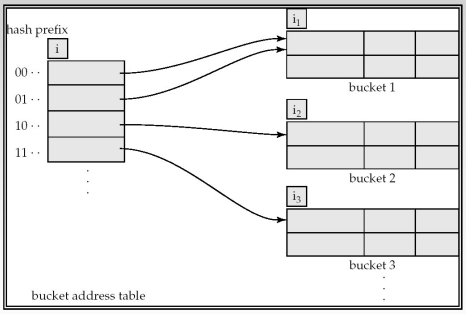
- Hash indices are always secondary indices

Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses
- Databases grow over time. If initial number of buckets is too small, performance will degrade due to too much overflows
- If file size at some point in the distance future is anticipated and 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 and
  - impossible in 7-24 databases

Dynamic Hashing

- Some of the problems of static hashing can be avoided by allowing the number of buckets to be modified dynamically
- Extendable hashing
  - Dynamic expansion/shrinking of address space
  - When buckets expand no need to move records of other buckets



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? (cost amortization)
- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified key value
  - If range queries are common, ordered indices are to be preferred

Multiple-Key Access

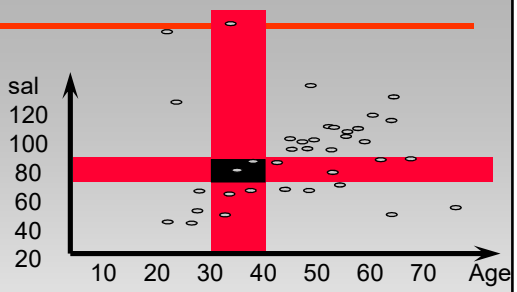
- Use multiple indices for certain types of queries.
- Example:

  select *account-number*
  from *account*
  where *branch-name* = "Perryridge" and *balance* < 1000

- Possible strategies for processing query using indices on single attributes:
  1. Use index on *branch-name* to find accounts with balances of $1000; test *branch-name* = "Perryridge".
  2. Use index on *balance* to find accounts with balances of $1000; test *branch-name* = "Perryridge".
  3. Use *branch-name* index to find pointers to all records to the Perryridge branch. Similarly find the pointers to *balance <1000*. Take intersection of both sets of pointers obtained.

Multi-Attribute Indexing

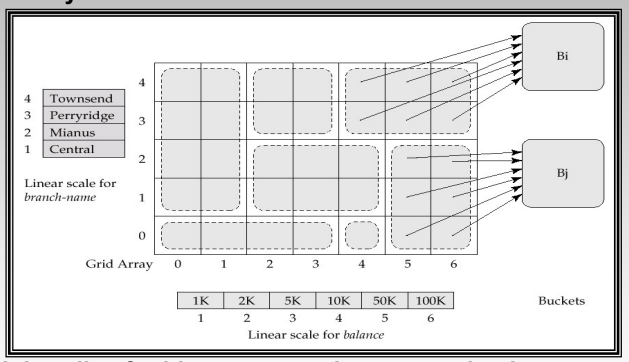
Suppose we want an index on a combined search-key (age,sal)
EMP(eno,ename,age,sal)



- separate indices- lots of false drops
- combined index based on composite key
  - key=sal*100+age (0 ≤ age ≤ 99)
  - search for 30 ≤ sal ≤ 40 → 3000 ≤ key ≤ 4000 fast
  - search for 60 ≤ age ≤ 80 → xx60 ≤ key ≤ yy80 slow (asymmetric)
- Grid files (2-d ISAM where the directory in MM)
- R-trees (2-d, 3-d, etc. B-trees)
- Quad trees, k-d-b-trees, and a number of trees, bushes, shrubs, etc.

Grid Files

- The grid file has a single grid array with one dimension for each search-key attribute



- 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

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
- 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

- A bitmap index on an attribute has a bitmap for each key te
  - Bitmap has as many bits as records (0 to n-1)
  - Bitmap index has one bitmap for each distinct value V of the attribute
  - For a attribute value V, the bits that correspond to the records that have the value V are 1 (on) the rest are 0 (off)

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

- Bitmap indices are useful for queries on multiple attributes
  - not particularly useful for single attribute queries
- Queries are answered using bitmap operations
  - Intersection 100110 AND 110011 = 100010
  - Union 100110 OR 110011 = 110111
  - Complementation NOT 100110 = 011001
- Examples:
  - Males with income level L1: 10010 AND 10100 = 10000

    Can then retrieve required tuples.

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

  - Counting number of matching tuples is even faster

Bitmap Indices

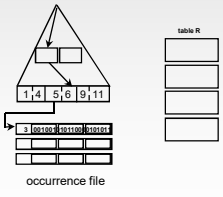
- Bitmap indices generally very small compared with relation size
  - E.g.
    - each record=100 bytes, Relation R=1m records, Relation size=100MB
    - 8 distinct values for an attribute
    - Bitmap = 1m bits / 8 bits/B = 125KB
    - Total size of the bitmap index = 8 distinct values * 125KB = 1MB
    - So total index size =1/100 of the relation (uncompressed)
    - If we compress the bitmaps it will be much less
- Deletion needs to be handled properly
  - Existence bitmap to note if there is a valid record at a record location
  - Needed for complement
    - not(*A=v*): *(NOT bitmap-A=v) AND ExistenceBitmap*
- Should keep bitmaps for all values including NULL
  - To handle SQL null semantics for NOT(*A=V*):
  - Take the bitmap for V and negate it
  - Take the bitmap of NULL and negate it
  - Then intersect the above two
- How about insertions?

FORGET THOSE

Efficient Implementation of Bitmap Operations

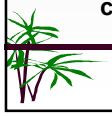
- Bitmaps are packed into words; a single word computes the "and" of 32 or 64 bits with 1 instruction
  - E.g. 1-million-bit maps can be and-ed with just 1,000,000/32=31,250 instructions
- Counting number of 1s can be done fast by a trick:
  - Use each byte to index into a precomputed array of 256 elements each storing the count of 1s in the binary representation
    - Can use pairs of bytes to speed up further at a higher memory cost
  - Add up the retrieved counts
- Bitmaps can be used instead of Tuple-ID lists at leaf levels of B$^+$-trees, for values that have a large number of matching records
  - Worthwhile if > 1/64 of the records have that value, assuming a tuple-id is 64 bits
  - Above technique merges benefits of bitmap and B$^+$-tree indices

00000000	0
00000001	1
…	
01011110	5
11111111	8



Index Definition in SQL

- **Create an index**
  create index <index-name> on <relation-name> (<attribute-list>)
  E.g.: create index *dept* on *department(dept_name)*
  create index teach on teaches(ID,semester)
- **Use create unique index to enforce the condition that the search key is a candidate key**
  - **Not really required if SQL unique integrity constraint in the create is supported**
- **To drop an index**
  drop index <index-name>
- **Most database systems allow specification of type of index, and clustering.**



289