CS 2550 / Spring 2006
Principles of Database Systems

06 – Indexing

Alexandros Labrinidis
University of Pittsburgh

Roadmap

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

Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute or 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

Indexing techniques evaluated on basis of:

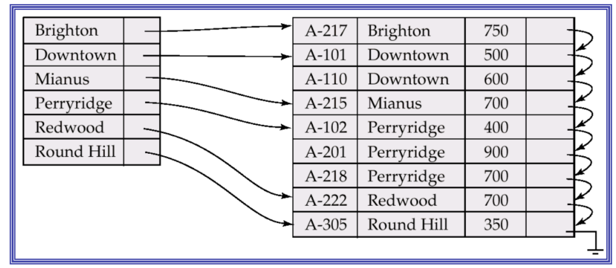
- Access types supported efficiently.
  For example:
  - records with a specified value in the attribute
  - records with an attribute value within a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead

Ordered Indices

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

Dense Index Files

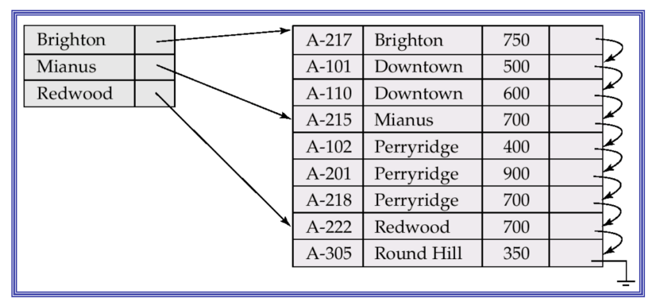
- Dense index — Index record appears for every search-key value in the file.



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 less than *K***
  - Search file sequentially starting at the record to which the index record points
- Advantages/disadvantages:
  - 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 of file, corresponding to least search-key value in the block.

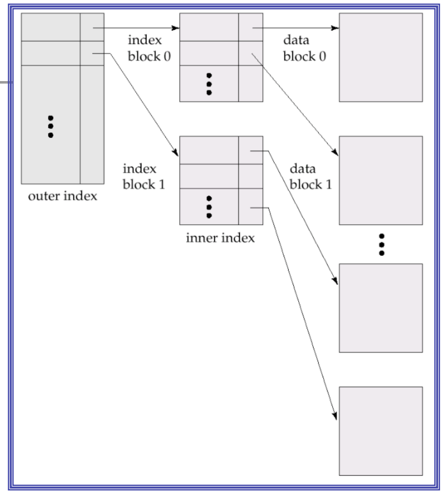
Example of Sparse Index Files



Multi-level Index

- If primary index does not fit in memory, access becomes expensive.
- To reduce number of disk accesses to index records, treat primary index kept on disk 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.
- Indices at all levels must be updated on insertion or deletion from the file.

Example



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.
- Single-level index deletion:
  - 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

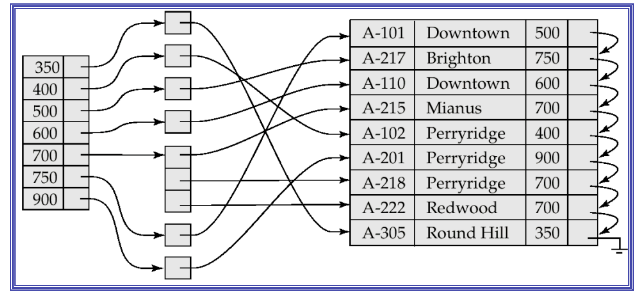
- Single-level index insertion:
  - Perform a lookup using the search-key value appearing in the record to be inserted.
  - Dense indices
    - if the search-key value does not appear in the index, insert it.
  - Sparse indices
    - if index stores an entry for each block of the file, no change needs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

Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field satisfy some condition (and the field is not the search-key of the primary index).
  - Example 1: In the *account* database stored sequentially by account number, we may want to find all accounts in a particular branch
  - Example 2: as above, but where we want to find all accounts with a specified balance or range of balances
- We can have 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 Example

- Secondary Index on ***balance*** field of ***account***



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 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

Roadmap

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

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.
- Disadvantage of B⁺-trees:
  - extra insertion and deletion overhead, space overhead.
- Advantages of B⁺-trees outweigh disadvantages
  - B⁺-trees are used extensively.

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 [n/2] and n children.
- A leaf node has between [(n–1)/2] 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	. . .	P_{n-1}	K_{n-1}	P_n

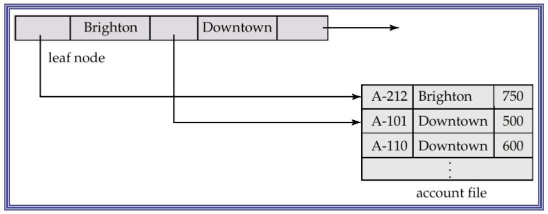
  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < . . . < K_{n-1}$$

Leaf Nodes in B⁺-Trees

- **Properties of a leaf node**
  - For i = 1, 2, . . ., n–1, 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 search-key does not form a primary key.
- If L_i, L_j are leaf nodes and $i < j$, L_i's search-key values are less than L_j's search-key values
- P_n points to next leaf node in search-key order

Leaf Node Example

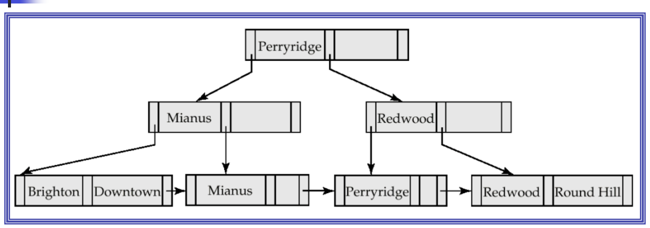


Non-Leaf Nodes in B^+-Trees

- 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_{m-1}

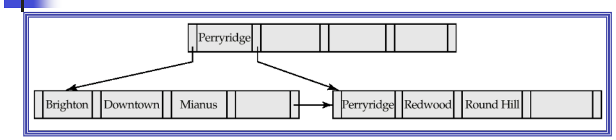
P_1	K_1	P_2	. . .	P_{n-1}	K_{n-1}	P_n

Example of a B^+-tree



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

Example of B^+-tree



B^+-tree for *account* file (n - 5)

- Leaf nodes must have between 2 and 4 values ($\lceil (n-1)/2 \rceil$ and $n-1$, with $n = 5$).
- Non-leaf nodes other than root must have between 3 and 5 children ($\lceil (n/2 \rceil$ and n with $n = 5$).
- 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 non-leaf levels of the B+-tree form a hierarchy of sparse indices.
- The B+-tree contains a relatively small number of levels (logarithmic in the size of the main file),
  - searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently
  - the index can be restructured in logarithmic time

Queries on B+-Trees

- Find all records with a search-key value of *k.*
  - Start with the root node
    - Examine the node for the smallest search-key value > *k.*
    - If such a value exists, assume it is K_j. Then follow P_i to the child node
    - Otherwise $k \geq K_{m-1}$, where there are *m* pointers in the node. Then follow P_m 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.

Queries on B+-Trees (Cont.)

- In processing a query, a path is traversed in the tree from the root to some leaf node.
- If *K* search-key values in the file
  - path is no longer than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- Example:
  - A node is generally the same size as a disk block, typically 4 kilobytes
  - *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 free 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!

Updates on B+-Trees: Insertion

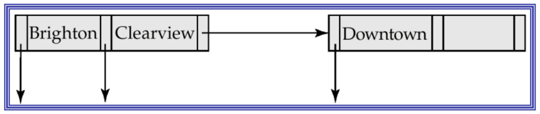
- Find the leaf node in which the search-key value would appear
- If the search-key value is already there (in leaf node),
  - record is added to file
  - if necessary, a pointer is inserted into the bucket.
- If the search-key value is not there
  - add the record to the main file and
  - create a bucket, if necessary.
  - 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) as discussed in the next slide.

Updates on B+-Trees: Insertion

- Splitting a node:
  - take the *n* (search-key value, pointer) pairs (including the one being inserted) in sorted order.
  - Place the first ⌈ *n*/2 ⌉ 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.

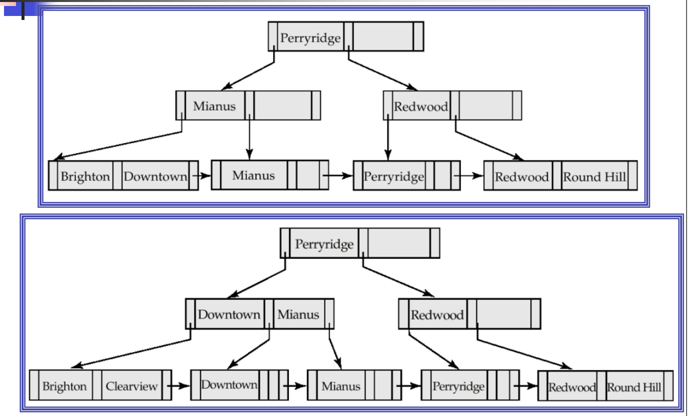
Updates on B+-Trees: Insertion

- The 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 Brighton and Downtown on inserting Clearview

Updates on B+-Trees: Insertion



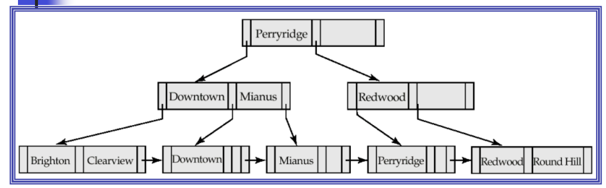
Updates on B+-Trees: Deletion

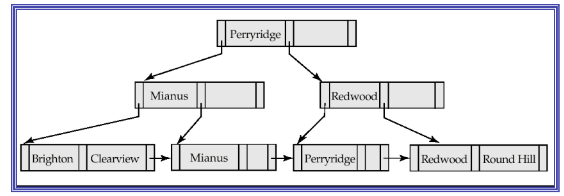
- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then
  - 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

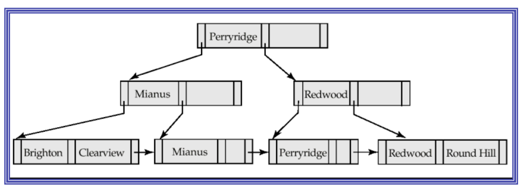
- Otherwise, if the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
  - Update the corresponding search-key value 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.

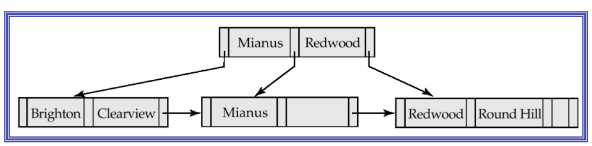
B$^+$-Tree Deletion / no cascade





B$^+$-Tree Deletion / cascade

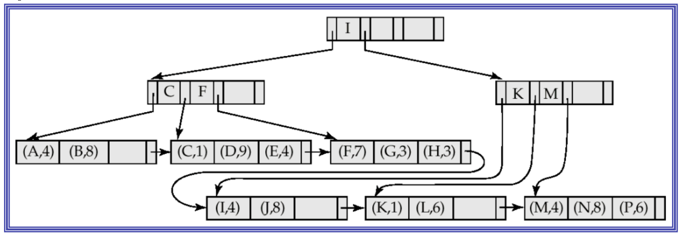




B$^+$-Tree File Organization

- Index file degradation problem is solved by using B$^+$-Tree indices.
- Data file degradation problem is solved by using B$^+$-Tree File Organization.
- The leaf nodes in a B$^+$-tree file organization store records, instead of pointers.
- 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 nonleaf node.
- Leaf nodes are still required to be half full.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B$^+$-tree index.

B⁺-Tree File Organization (Cont.)



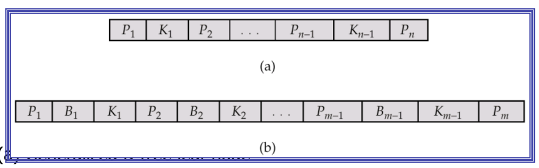
- Good space utilization important since records use more space than pointers.
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
  - Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least $\lfloor 2n/3 \rfloor$ entries

Roadmap

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

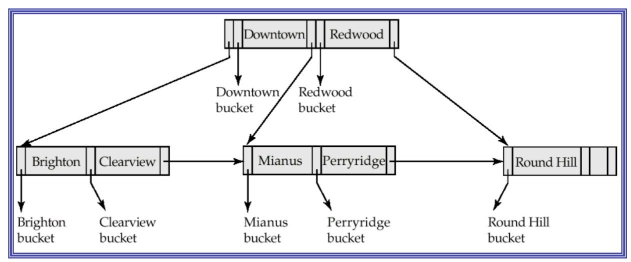
B-Tree Index Files

- 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.

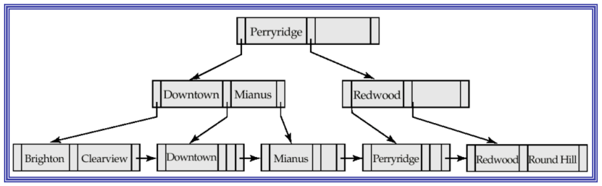


- (a)
- (b) Nonleaf node – pointers B_i are bucket /file record pointers

B-Tree Index File Example



B^+-Tree Index File Example



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.
  - 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.

Roadmap

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

Static Hashing

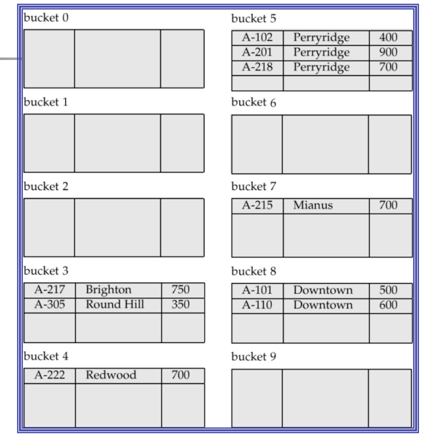
- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash function.**
- Hash function *h* is a function from the set of all search-key values *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
  - entire bucket has to be searched sequentially to locate a record.

Static Hashing – Examples

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

- 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) = 5 h(Round Hill) = 3 h(Brighton) = 3

Example



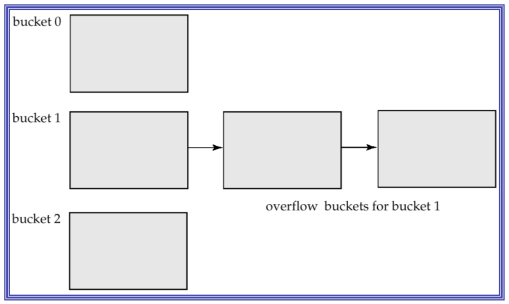
Hash Functions

- 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**
  - each bucket is assigned the same number of search-key values from the set of *all* possible values.
- Ideal hash function is **random**
  - 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.*
- 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.

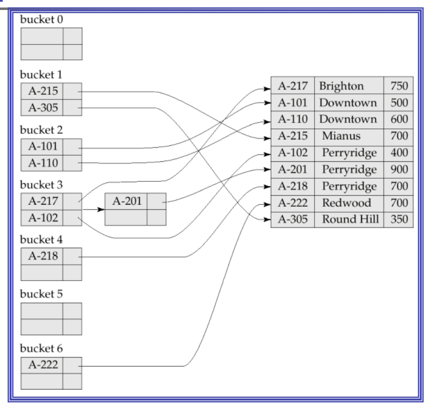
Overflow chaining example



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 with 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 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.
- These problems can be avoided by using techniques that allow the number of buckets to be modified dynamically.

Roadmap

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

Ordered Indexing vs 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

Index Definition in SQL

- Create an index

  create 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 is a candidate key.
  - Not really required if SQL **unique** integrity constraint is supported
- To drop an index

  drop index <index-name>