Database Systems

Physical Representation: Indexing and Hashing

Prof. Dr. Agnès Voisard   Muhammed-Ugur Karagülle

Institute of Computer Science, Databases and Information Systems Group

Fraunhofer FOKUS

2025

Notes

Notes

Outline

Access Methods

- **Access methods** are a group of programs that allow operations to be applied to a file
  - We can apply several access methods to a file organization
  - Some require the use of an index
- All the records in a file are not necessarily accessed:
  - E.g.: find all accounts in *Perryridge* branch
    $\Rightarrow$ inefficient to read every record (sequential search)
  - Idea: access the records directly
- **Two general approaches**:
  - Indices
  - Hash functions

- **Indexing mechanisms** used to speed up access to desired data
  - E.g., branch_name in bank application
- **Index:** Data structure which allows to locate information faster than with sequential scan
- **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:

$$(SearchKey, Pointer)$$

- Index files are typically **much smaller** than the original file

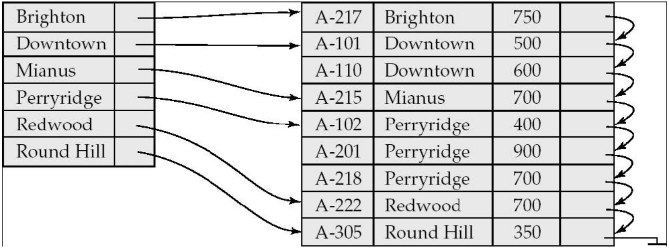
- Two basic kinds of indices:
  - **Ordered Indices**:
    - Search keys are stored sorted on the search key value, e.g., author catalog in library
  - **Hash indices**:
    - Search keys are distributed uniformly across *buckets* using a *hash function*

- Efficiency of accessed types. E.g.,
  - Records with a specified value in the attribute
  - Or records with an attribute value falling in a specified range of values (e.g., 10.000 < Amount< 40.000)
- Access time
- Insertion time
- Deletion time
- Space overhead

- **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 primary index

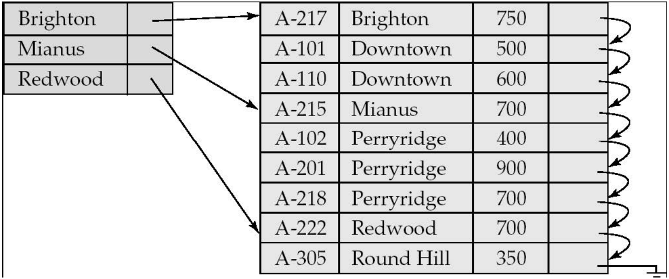
Dense Index Files

- **Dense index**: Index record appears for **every search-key value** in the file



Sparse index File

- **Sparse index**: Contains index records **for only some search-key values**
  - Applicable when records are sequentially ordered on search-key



Sparse index File (cont'd)

- To locate a record with search-key value *K*:
  - Find index record with **largest search-key value < *K***
  - Search file sequentially starting at the record to which the index record points

Example:

E.g., find the records of the *Perryridge* branch

- Last entry before *Perryridge* is *Mianus*
- Follow that pointer until the first *Perryridge* record is found

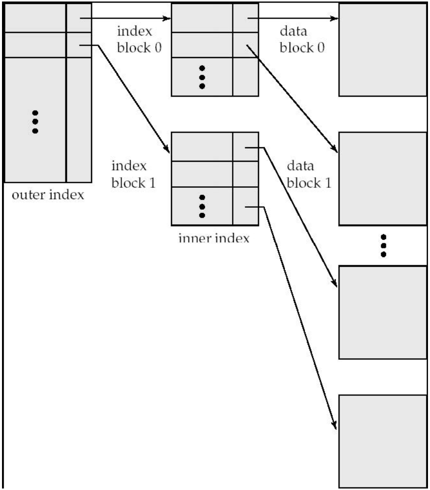
- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions
  - Generally slower than dense index for locating records

Dense vs. Sparse Indices

- Dense index: Direct access, faster to locate a record than sparse index
- Sparse index: Less space and easier to maintain
- **Trade-off access time and space overhead**
- **Good trade-off**: Sparse index with one index entry per block, corresponding to least search-key value in the block

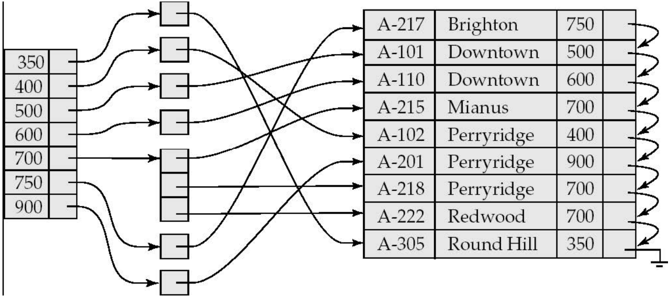
- If primary index does not fit in memory, access becomes expensive.
- Solution: Treat **primary index** kept on disc as a **sequential file** and **construct a sparse index** on it
  - Outer index: A sparse index of primary index
  - Inner index: The primary index file
- If even outer index is too large 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

Multilevel Index (cont'd)



- Index whose search key specifies an **order different from the sequential order** of the file (non-clustering index)
- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value

- Secondary indices: dense or sparse



- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value

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 disc
  - Block fetch requires about 5 to 10 micro seconds, versus about 100 nanoseconds for memory access

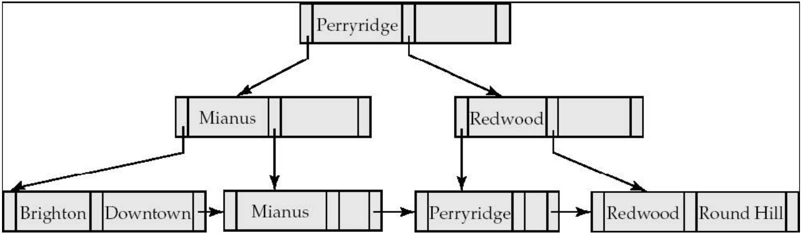
- B^+-trees indices are an alternative to indexed-sequential files.

- **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^+ trees**:
  - Automatically reorganizes itself with small, local, changes, when insertions/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

B^+-Tree Index Files (cont'd)

- 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 $\lceil \frac{n}{2} \rceil$ and n children, n branching factor
  - A leaf node has between $\lceil \frac{n-1}{2} \rceil$ and $n-1$ values

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



- Use multiple indices for certain types of queries

Example:

SELECT *account_number*
FROM *account*
WHERE *branch_name* = ”Perryridge” AND *balance* = 100

- Possible strategies for processing query using indices on single attributes:
  1. Use index on *branch_name* to find accounts with branch name ”Perryridge”; test *balance* = 100
  2. Use index on *balance* to find accounts with balances of 100; test *branch_name* = ”Perryridge”
  3. Use branch_name index to find pointers to all records pertaining to the ”Perriridge” branch. Similarly use index on balance. Take intersection of both sets of pointers obtained.

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

- Suppose we have an index on combined search-key (*branch_name*, *balance*)
- For ”WHERE branch_name = ’Perryridge’ AND balance = 100” the index on (branch_name, balance) can be used to fetch only records that satisfy both conditions
  - Using separate indices in less efficient - we may fetch many records (or pointers) that satisfy only one of the conditions
- Can also efficiently handle ”WHERE branch_name = ’Perryridge’ AND balance < 100”
- But can not efficiently handle ”WHERE branch_name < ’Perryridge’ AND balance = 100”
  - May fetch many records that satisfy the first but not the second condition

- 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 his 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 *account* file, using *branch_name* as key
  - There are 10 buckets,
  - The binary representation of the *i*-th character is assumed to be integer *i*.
  - The hash function returns the sum of the binary representations of the characters modulo 10
    - $h(Perryridge) = 5$
    - $h(RoundHill) = 3$
    - $h(Brighton) = 3$

bucket 0

bucket 1

bucket 2

bucket 3

A-217	Brighton	750
A-305	Round Hill	350

bucket 4

A-222	Redwood	700

bucket 5

A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700

bucket 6

bucket 7

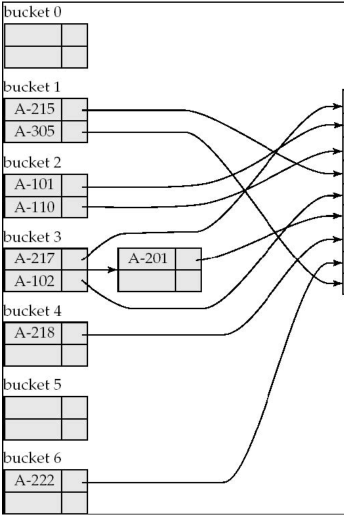
A-215	Mianus	700

bucket 8

A-101	Downtown	500
A-110	Downtown	600

bucket 9

Example of Hash Index



A-217	Brighton	750
A-101	Downtown	500
A-110	Downtown	600
A-215	Mianus	700
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

- 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+-treesCost of periodic re-organization

Index Definition in SQL

- Create an index:
  CREATE INDEX *index_name* **ON** *relation_name* (*attribute_list*)

Example:

CREATE INDEX *b-index* **ON** *branch*(*branch_name*)

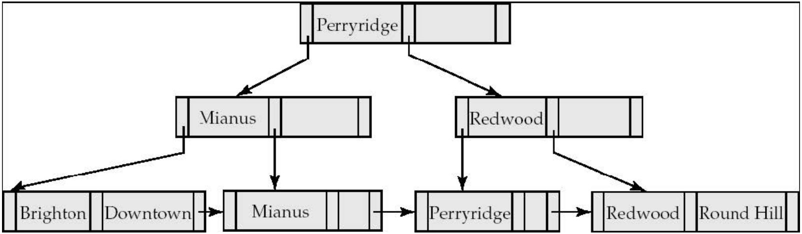
- Use CREATE UNIQUE INDEX to indirectly specify and enforce the condition that the search-key is a candidate key.
  - Not really required if SQL unique integrity constraint is supported
- To drop an index: **DROP INDEX** *index_name*
- Most database systems allow specification of type of index, and clustering.

Questions?

What will come next?

1. Welcome to Database Systems
2. Introduction to Database Systems
3. Entity Relationship Design Diagram (ERM)
4. Relational Model
5. Relational Algebra
6. Structured Query Language (SQL)
7. Relational Database Design - Functional Dependencies
8. Relational Database Design - Normalization
9. Online Analytical Processing + Embedded SQL
10. Data Mining
11. Physical Representation - Storage and File Structure
12. Physical Representation - Indexing and Hashing
13. Transactions
14. Concurrency Control Techniques
15. Recovery Techniques
16. Query Processing and Optimization

- Special cases:
  - If a root is not a leaf, it has at least two children
  - If a 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

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

Leaf Nodes in B^+ Trees

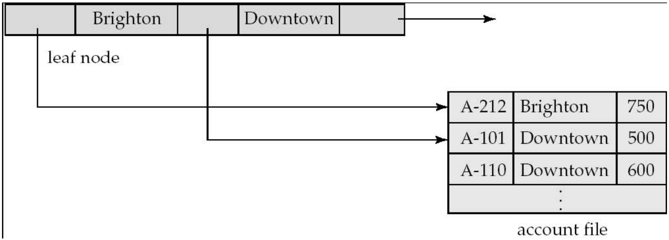
Properties of a leaf node:

- For $i = 1, 2, \ldots, n-1$, pointer P_i
  - Either **pointers to a file record** with search-key value K_i,
  - Or **to 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

Leaf Nodes in B^+-Trees (cont'd)

Properties of a leaf node:

- P_n points to next leaf node in search-key order

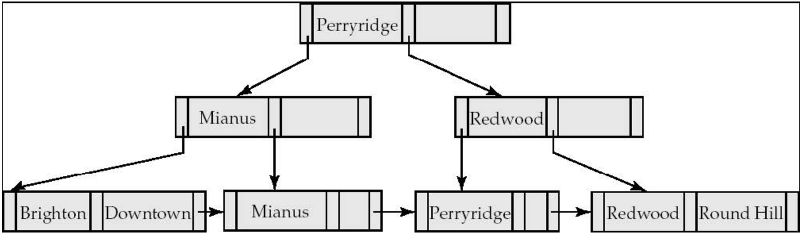


- **Non-leaf nodes** form **a multi-level sparse index** on the leaf nodes. For a non-leaf node with n pointers:
  - All search-keys in the subtree to which P_1 points are **less than** K_1
  - For $2 \leq i \leq n-1$, all the search-keys in the subtree to which P_i points have **values greater than 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}

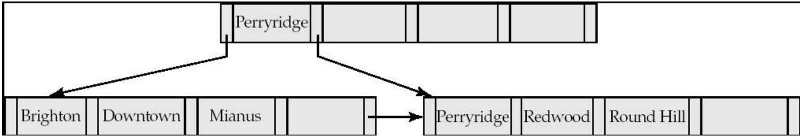
P_1	K_1	P_2	$\ldots$	P_{n-1}	K_{n-1}	P_n

B^+ Tree: Example

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



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



- Leaf nodes must have between 2 and 4 values
  - $\lceil \frac{n-1}{2} \rceil$ and $n - 1$, with $n = 5$
- Non-leaf nodes other than root must have between 3 and 5 children
  - $\lceil \frac{n}{2} \rceil$ 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**
- B^+ tree contains a relatively small number of levels
  - Level below root has $2 \cdot \lceil \frac{n}{2} \rceil$ values
  - Next level has at least $2 \cdot \lceil \frac{n}{2} \rceil \cdot \lceil \frac{n}{2} \rceil$ values
  - …
  - If there are K search-key values in the file, the tree height is no more than $\lceil log_{\lceil \frac{n}{2} \rceil}(K) \rceil$
  - Thus searches can be conducted efficiently
- Insertions ad deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time

- Lookup: Find all records with a search-key value of k

  1 $N = root$

  2 Repeat:

    2.1 Examine N for the smallest search-key value > k

    2.2 If such a value exists

      2.2.1 Assume it is K_i

      2.2.2 Set $N = P_i$

    2.3 Otherwise $k \geq K_{n-1}$

      2.3.1 Set $N = P_n$

    Until N is a leaf node

  3 If for some i, key $K_i = k$ follow pointer P_i to the desired record or bucket

  4 Else no record with search-key value k exists

- If there are K search-key values in the file, the height of the tree is no more than $\lceil log_{\lceil \frac{n}{2} \rceil}(K) \rceil$
- **A node is generally the same size as a disc 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 $log_{50}(1000000) = 4$ 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 (every node access may need a disc I/O, costing around 20 milliseconds)

Updates on B^+-Trees: Insertion

1 Find the leaf node in which the search-key value would appear (lookup)

2 **If the search-key value is already present in the leaf node**

  2.1 Add record to the file

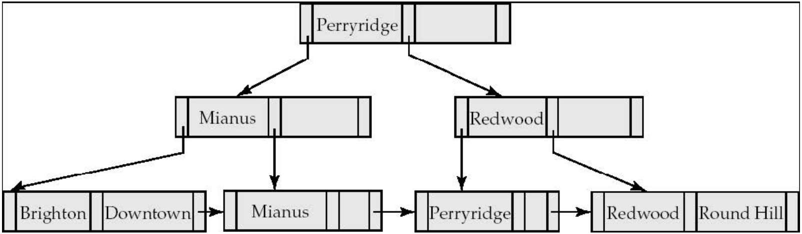
3 **If the search-key value is not present**, then

  3.1 Add the record to the main file (and create a bucket if necessary)

  3.2 If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node

  3.3 Otherwise, split the node (along with the new (key-value,pointer) entry)

Example of a B^+-Tree



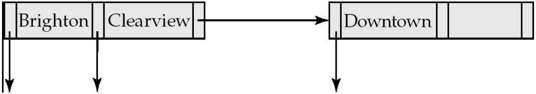
- B^+ tree for account file ($n = 3$)

Updates on B^+-Trees: Insertion

- Splitting a leaf node:
  - Take the *n* (search-key values, pointer) pairs (including the one being inserted) in sorted order
    Place the first $\lceil \frac{n}{2} \rceil$ **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

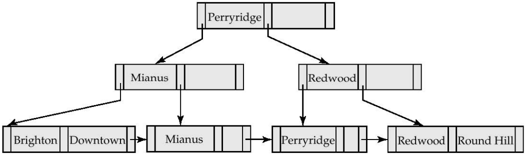
Updates on B^+-Trees: Insertion (cont'd)

- 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

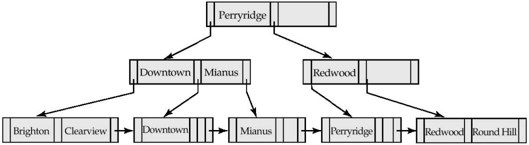


  - Result of splitting node containing Brighton and Downtown on **inserting "Clearview"**
  - Next step: insert entry with (Downtown,pointer-to-new-node) into parent

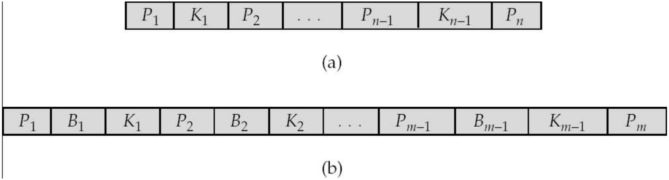
Updates on B^+-Trees: Insertion (cont'd)



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

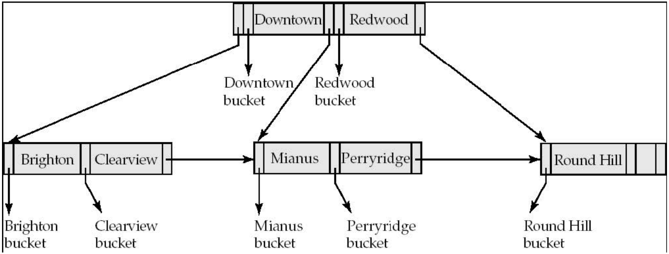


- Similar to B^+ tree, but B tree allows search-key values to appear only once
  $\Rightarrow$ eliminates redundant storage of search-keys
- Search-keys in non-leaf nodes appear nowhere else in the B tree; an additional pointer field for search-key in a non-leaf node must be included
  - Generalized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

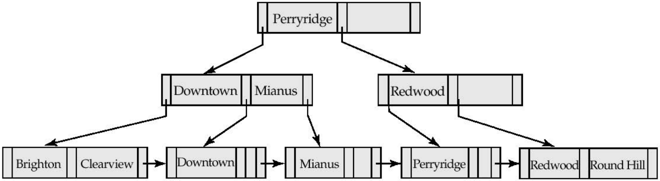


- Non-leaf node - pointers B_i are the bucket or file record pointers

Example of *B*-Tree and B^+-Tree Index File



- *B* tree (above) and B^+ tree (below) 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 values 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*-rees typically have greater depth than corresponding B^+ trees
  - Insertion and deletion more complicated than in B^+-Trees
  - Implementation harder than B^+-Trees
- Typically, advantages of *B* trees do not out weight disadvantages

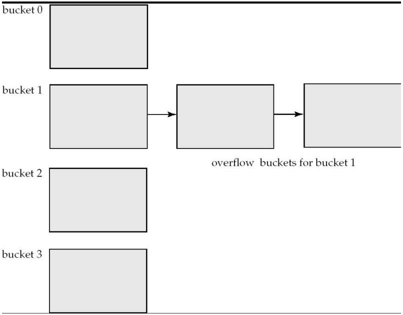
- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of all possible values.
- Ideal 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'd)

- **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.

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.