Chapter 11 Indexing and Hashing

Two kinds of indices:

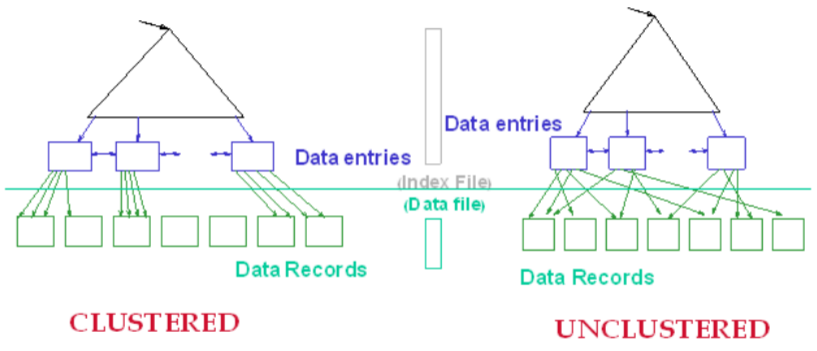
1. Ordered indices. Based on sorted ordering of the values.
2. Hash indices. Based on uniform distribution of value across a range of buckets. The buckets to which a value is assigned is determined by a function, called *hash function*.

Evaluation of different indexing techniques:

- Access types
- Access time
- Insertion time
- Deletion time
- Space overhead

I. Ordered Indices

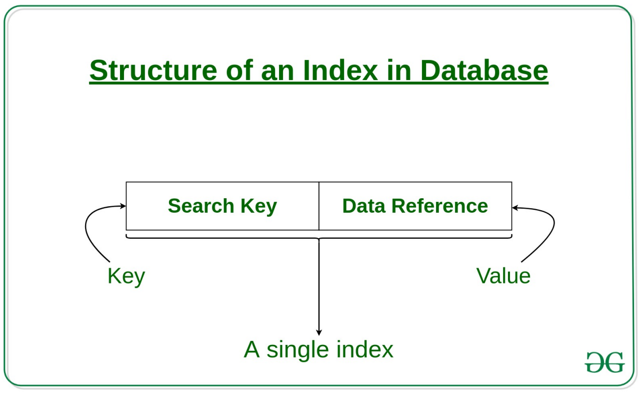
1. Indices Classification - Clustered vs. Nonclustering
   - If the file containing the records is sequentially ordered, a ***clustered index*** is an index whose search key is also defines the sequential order of the file. With a clustered index the rows are stored physically on the disk in the same order as the index. Therefore, there can be only one clustered index.
   - Indices whose search key specifies an order different from the sequential order of the file are called ***nonclustering indices***, or ***secondary indices***. With a non clustered index there is a second list that has pointers to the physical rows. You can have many non clustered indices, although each new index will increase the time it takes to write new records.



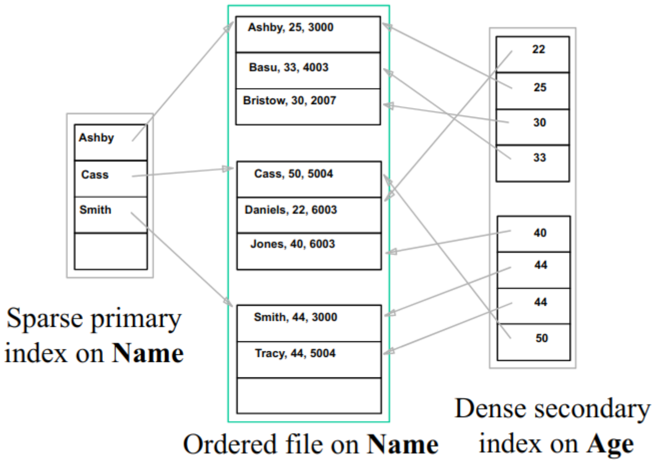
   Note: Files, with a clustering index on the search key, are called ***index-sequential files***

2. Dense vs. Sparse

   An ***index entry***, or ***index record***, consists of a search key value and pointers to one or more records with that value as their search-key value.



- ***Dense Index***: An index entry appears for every search-key value in the file
- ***Sparse Index***: An index entry appears for only some of the search-key values. Sparse indices can be used only if the relation is stored in sorted order of the search key, that is, if the index is a clustered index.



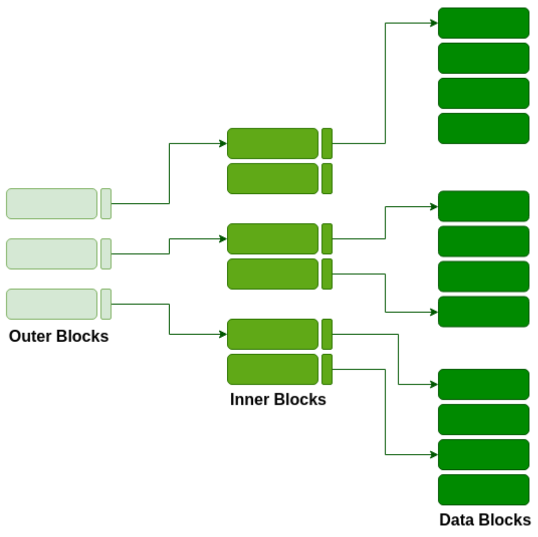
In general, dense indices are faster than sparse indices, but there's a trade-off between speed and space.

3. Multievel Indices

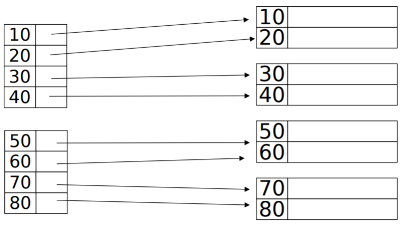
With the growth of the size of the database, indices also grow. As the index is stored in the main memory, a single-level index might become too large a size to store with multiple disk accesses.

The multilevel indexing segregates the main block into various smaller blocks so that the same can stored in a single block.

The *outer blocks* are divided into *inner blocks* which in turn are pointed to the data blocks. This can be easily stored in the main memory with fewer overheads.

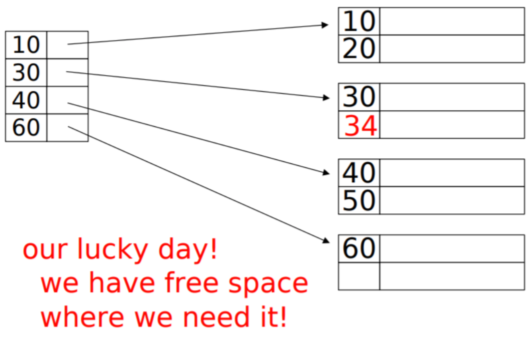


4. Index Update

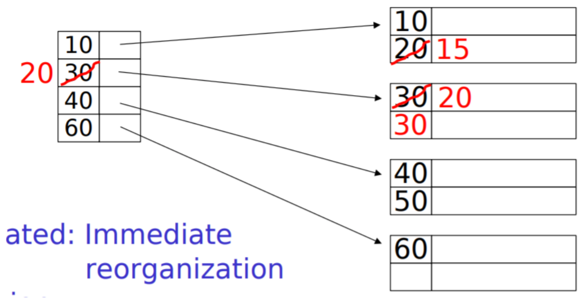


1. Insertion
   - Dense Indices
     - Search-key value not in the index. Find an appropriate position and insert.
     - Otherwise
       - If the index entry stores pointers to all records with the same search-key value, the system adds a pointer to the new record in the index entry.
       - If the index entry stores a pointer to only the first record with the search-key value. The system then places the record being inserted after the other recorders with the same search-key values.
   - Sparse Indices
     - We assume that the index stores an entry for each block. If the system creates a new block, it inserts the first search-key value (in search-key order) appearing in the new block into the index.
     - On the other hand, if the new record has the least search-key value in its block, the system updates the index entry pointing to the block
     - If not, the system makes no change to the index.

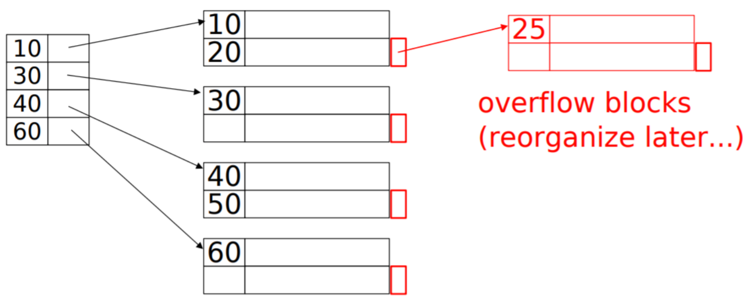
   Example: Insert 34(suppose there's a space between 30 and 40)



   Example: Insert 15

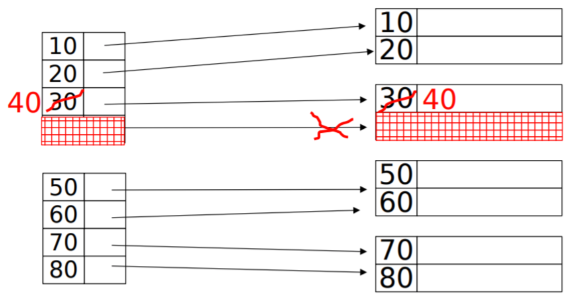


   Example: Inset 25



2. Deletion
    - Dense Indices
        - If the deleted record is a unique search-key value, then the system deletes the corresponding index entry from the index.
        - Otherwise
            - If the index entry stores pointers to all records with the same search-key value, the system deletes the pointer to the deleted the pointer to the deleted record from the index entry.
            - If the index entry stores a pointer to only the first record with the search-key value, the system update the index entry to point to the next record.

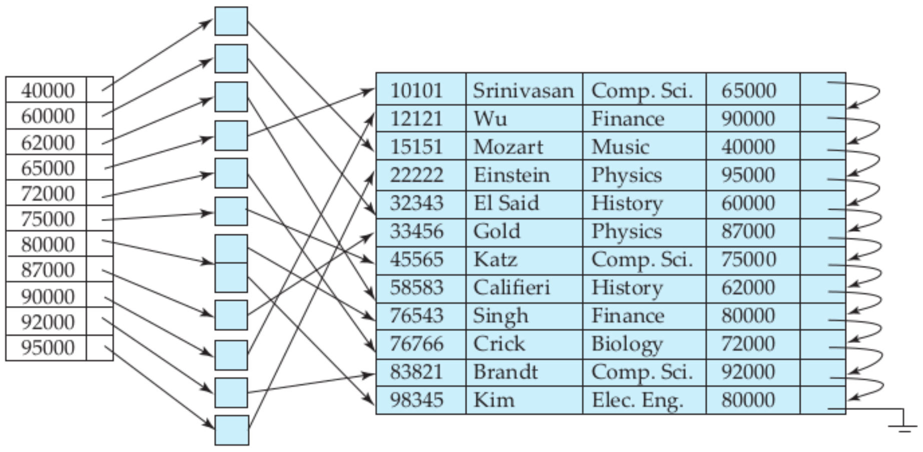
    Example: Delete 30



    - Sparse Indices
        - If the index does not contain an index entry with the search-key value of the deleted record, nothing needs to be done to the index.
        - Otherwise
            - If the deleted record was the only record with its search key, the system replaces the corresponding index record with an index record for the next search-key value(in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.
        - If the index entry for the search-key value points to the record being deleted, the system updates the index entry to point to the next record with the same search-key value.

5. Secondary Indices
    - Secondary indices must be dense, with an index entry for every search-key value, and a pointer to every record in the file.
        - If a secondary index stores only some of the search-key values, records with intermediate search-key values may be anywhere in the file and, in general, we cannot find them without searching the entire file.
    - With secondary indices:

        - Lowest level is dense
        - Other levels are sparse



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
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

Figure 11.6 Secondary index on *instructor* file, on noncandidate key *salary*.

6. Indices on Multiple Keys
    - A search key containing more than one attribute is referred to as a ***composite search key***.
    - The search key can be represented as a tuple of values, of the form (a 1 , . . . , a n), where the indexed attributes are A_1 , . . . , A_n .
    - The ordering of search-key values is the ***lexicographic ordering***.
        - Example: for the case of two attribute search keys, (a_1 , a_2) < (b_1 , b_2) if either a_1 < b_1 or a_1 = b_1 and a_2 < b_2

II. 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, and they are used extensively.

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 ⌈ n/2 ⌉ children.
- A leaf node contains as few as ⌈(n–1)/2⌉ 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.

---

1. Structure of a B + -Tree
    - Typical Node

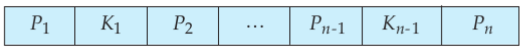
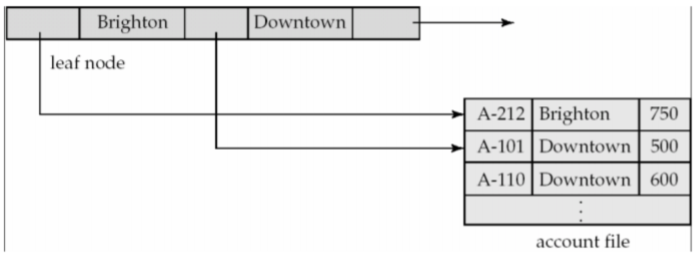
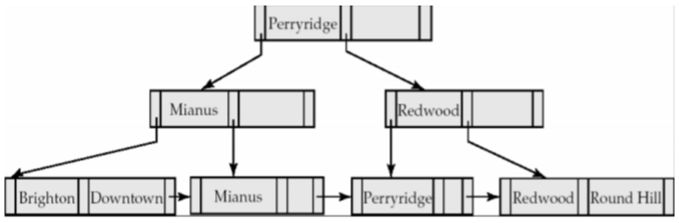


Figure 11.7 Typical node of a B⁺-tree.

        - 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 $K_1 < K_2 < K_3 < \ldots < K_{n-1}$
    - Leaf Nodes in B^+ -Trees
        - 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



    - 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 ≤ i ≤ 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}



B⁺-tree for *account* file (*n* = 3)

2. Queries on B + -Trees

    Find all records with a search-key value of k.

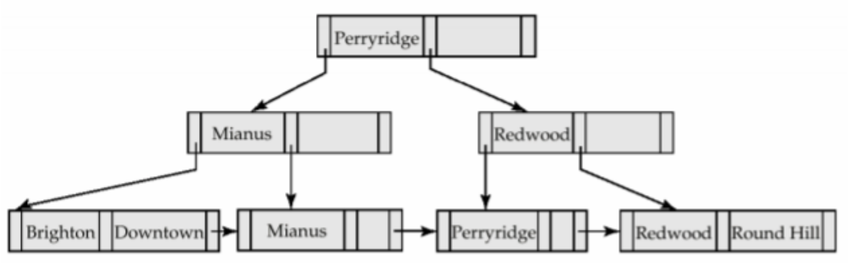
    1. Start with the root node
        1. Examine the node for the smallest search-key value > k.
        2. If such a value exists, assume it is K_j. Then follow P_i to the child node
        3. Otherwise $k \geq K_{m-1}$, where there are m pointers in the node. Then follow P_m to the child node.
    2. If the node reached by following the pointer above is not a leaf node, repeat step 1 on the node
    3. Else we have reached a leaf node.

1. If for some i, key $K_i = k$ follow pointer P_i to the desired record or bucket.
2. Else no record with search-key value k exists.

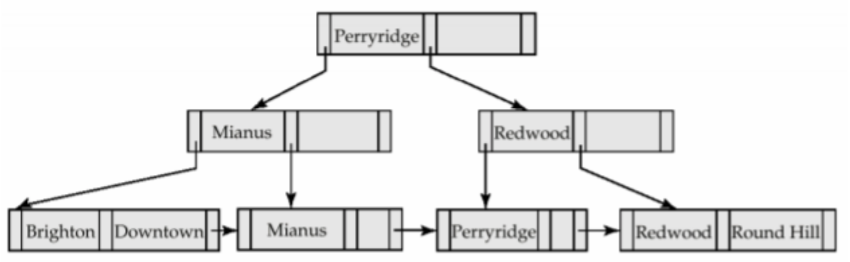
3. Updates on B + -Trees

   1. Insertion

      - Find the leaf node in which the search-key value would appear
      - If the search-key value is already there in the leaf node, record is added to file and if necessary a pointer is inserted into the bucket.
      - If the search-key value is not there, then add the record to the main file and create a bucket if necessary. Then:
        - If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
        - Otherwise, split the node (along with the new (key-value, pointer) entry)
      - 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.



Insert "Clearview" into the tree.

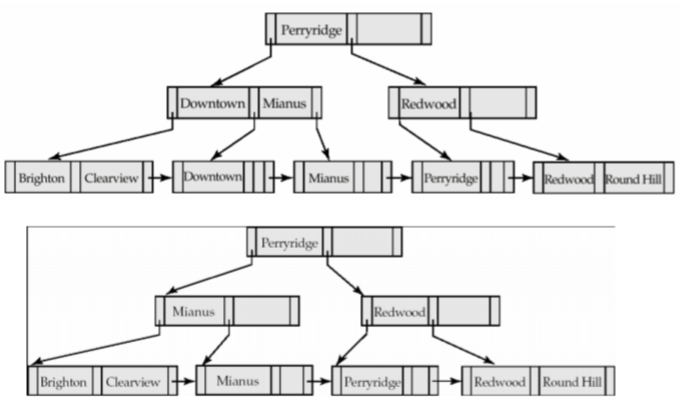


   2. Deletion

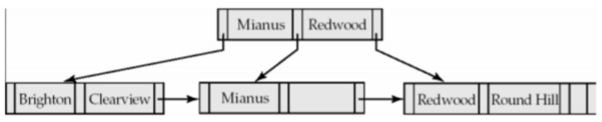
      - Find the record to be deleted, and remove it from the main file and from the bucket (if present)
      - Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
      - If the node has too few entries due to the removal, and the entries in the node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

        - 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 ⌈ n/2 ⌉ or more pointers is found. If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

Examples of B+-Tree Deletion



Before and after deleting "Downtown"



Deletion of "Perryridge" from result of previous example

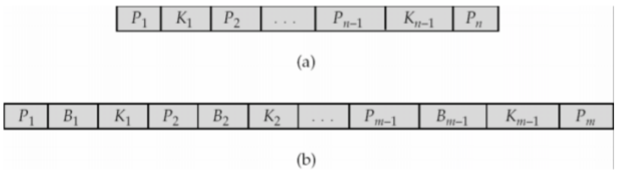
4. Nonunique Search Keys
    - One problem with nonunique search keys is in the efficiency of record deletion.
        - Suppose a particular search-key value occurs a large number of times, and one of the records with that search key is to be deleted. The deletion may have to *search through a number of entries*, potentially across multiple leaf nodes, to find the entry corresponding to the particular record being deleted.
    - A simple solution is to make search keys unique by creating a composite search key containing the original search key and another attribute, which together are unique across all records.
    - An alternative is to store each key value only once in the tree, and to keep a bucket (or list) of record pointers with a search-key value, to handle nonunique search keys. This approach is more space efficient since it stores the key value only once; however, it creates several complications when B + -trees are implemented.
5. Complexity of B + -Tree Updates
    - Insertion time complexity: `O(logn)`
    - Deletion time complexity: `O(logn)`
    - Space complexity: `O(n)`

III. B + -Tree Extensions

1. B + -Tree File Organization

- In a B + -tree file organization, the leaf nodes of the tree store records, instead of storing pointers to records.
- Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a 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.
- 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 ⌊2n / 3⌋ entries

2. Secondary Indices and Record Relocation
3. Indexing Strings
   - Variable length strings as keys
     - Variable fanout
     - Use space utilization as criterion for splitting, not number of pointers
   - Prefix compression
     - Key values at internal nodes can be prefixes of full key
       - Keep enough characters to distinguish entries in the subtrees separated by the key value
       - E.g. "Silas" and "Silberschatz" can be separated by "Silb"
4. Bulk Loading of B + -Tree Indices
5. 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


P_1 K_1 P_2 ... P_{n-1} K_{n-1} P_n
(a)
P_1 B_1 K_1 P_2 B_2 K_2 ... P_{m-1} B_{m-1} K_{m-1} P_m
(b)

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

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 out weigh disadvantages.

6. Flash Memory

IV. Multiple-Key Access

Consider the query: “Find all instructors in the Finance department with salary equal to $80,000.”

1. Using Multiple Single-Key Indices

   ```
   select ID
   from instructor
   where dept name =’Finance’ and salary= 80000;
   ```

   Three strategies possible for processing this query:

   1. Use the index on dept name to find all records pertaining to the Finance department. Examine each such record to see whether salary= 80000.
   2. Use the index on salary to find all records pertaining to instructors with salary of $80,000. Examine each such record to see whether the department name is “Finance”.
   3. Use the index on dept name to find pointers to all records pertaining to the Finance department. Also, use the index on salary to find pointers to all records pertaining to instructors with a salary of $80,000. Take the intersection of these two sets of pointers. Those pointers that are in the intersection point to records pertaining to instructors of the Finance department and with salary of $80,000.
2. Indices on Multiple Keys
   - Use an index on combined search-key:
     - will 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

       ```
       select ID
       from instructor
       where dept name = ’Finance’ and salary< 80000;
       ```

     - But cannot efficiently handle

       ```
       select ID
       from instructor
       where dept name < ’Finance’ and salary< 80000;
       ```

       - May fetch many records that satisfy the first but not the second condition.

V. Static Hashing

File organizations based on the technique of hashing allow us to avoid accessing an index structure. Hashing also provides a way of constructing indices.

Bucket - a unit of storage that can store one or more records. A bucket is typically a disk block, but could be chosen to be smaller or larger than a disk block.

To insert a record with search key K_i , we compute $h(K_i)$, which gives the address of the bucket for that record.

Hashing can be used for two different purposes.

1. In a **hash file organization**, we obtain the address of the disk block containing a desired record directly by computing a function on the search-key value of the record.
2. In a **hash index organization** we organize the search keys, with their associated pointers, into a hash file structure.

> In static hashing, when a search-key value is provided, the hash function always computes the same address.

---

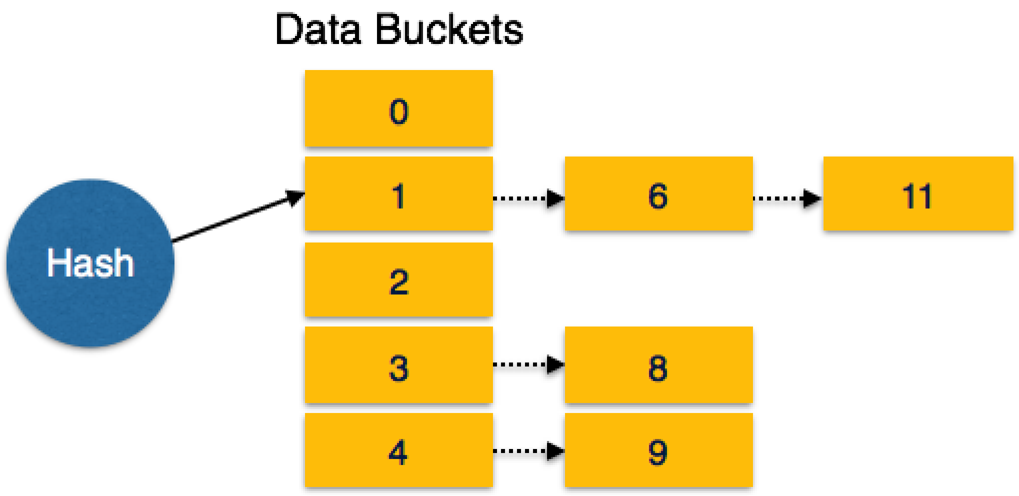
1. Hash Function

   An ideal hash function distributes the stored keys uniformly across all the buckets, so that every bucket has the same number of records (minimize the conflict).

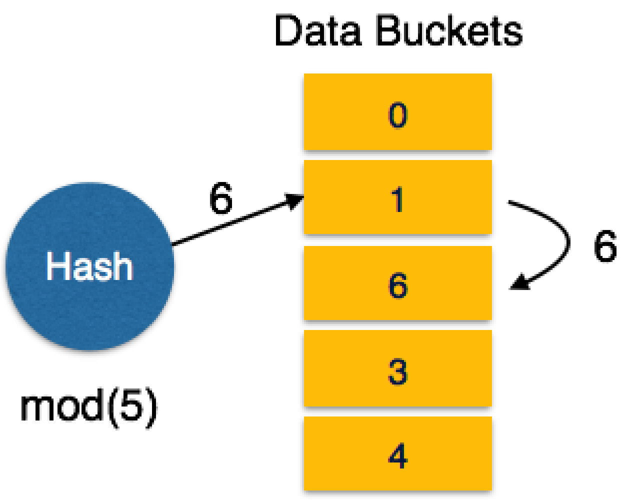
   - Uniform Distribution
   - Random Distribution
2. Handling of Bucket Overflows(collision)
   - Bucket Overflows - the problem

     If the bucket does not have enough space, a bucket overflow is said to occur. Bucket overflow can occur for several reasons:

     - **Insufficient Buckets**. Number of buckets is predefined but not enough.
     - **Skew**. Some buckets are assigned more records than are others, so a bucket may overflow even when other buckets still have space. This can occur for two reasons:
       1. Multiple records may have the same search key.
       2. The chosen hash function may result in nonuniform distribution of search keys.
   - Overflow Bucket - the solution
     - **Overflow Chaining** – When buckets are full, a new bucket is allocated for the same hash result and is linked after the previous one. This mechanism is called **Closed Hashing**.



     - **Linear Probing** – When a hash function generates an address at which data is already stored, the next free bucket is allocated to it. This mechanism is called **Open Hashing**.



- Drawback

  We must choose the hash function when we implement the system, and it cannot be changed easily thereafter if the file being indexed grows or shrinks.

3. Hash Indices
   - A hash index organizes the search keys, with their associated pointers, into a hash file structure.
   - We apply a hash function on a search key to identify a bucket, and store the key and its associated pointers in the bucket (or in overflow buckets).
   - Strictly speaking, hash indices are only secondary index structures.

VI. Dynamic Hashing

If we are to use **static hashing** for a database that grow larger over time, we have three classes of options:

1. Choose a hash function based on the current file size. This option will result in performance degradation as the database grows.
2. Choose a hash function based on the anticipated size of the file at some point in the future. Although performance degradation is avoided, a significant amount of space may be wasted initially.
3. Periodically reorganize the hash structure in response to file growth. Such a reorganization involves choosing a new hash function, recomputing the hash function on every record in the file, and generating new bucket assignments.

Dynamic Hashing techniques allow the hash function to be modifies dynamically to accommodate the growth or shrinkage of the database.

---

1. Data Structure

   Hash function, in dynamic hashing, is made to produce a large number of values and only a few are used initially.

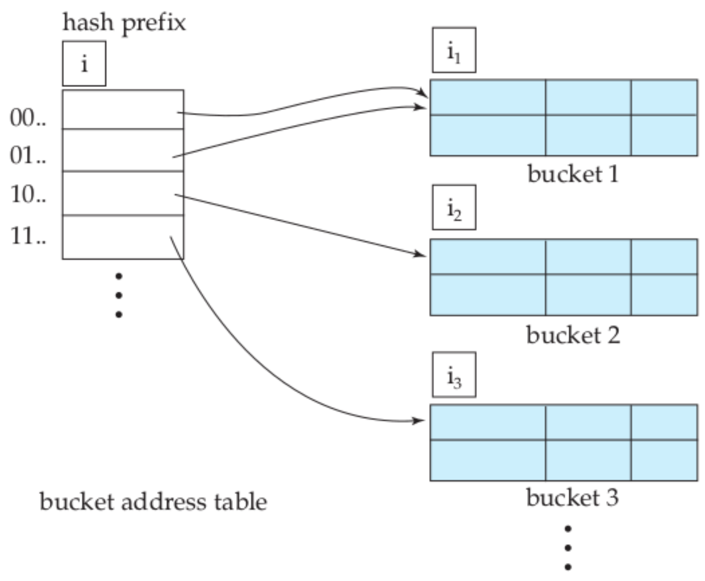


Figure 11.26 General extendable hash structure.

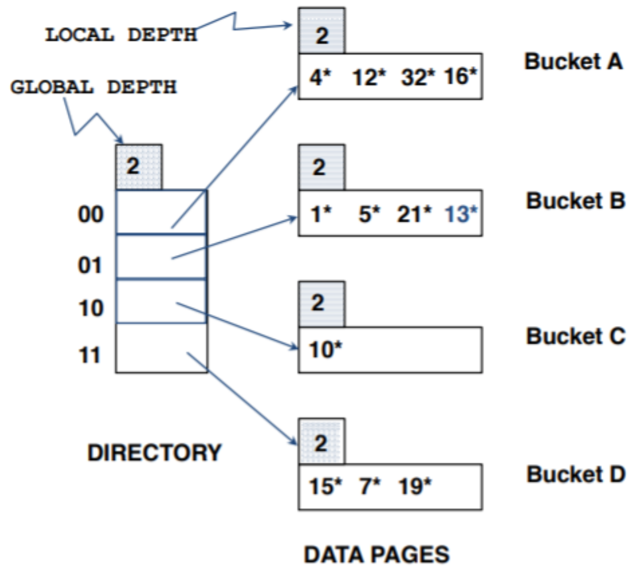
The *i* appearing above the bucket address table in the figure indicates that *i* bits of the hash value h(K) are required to determine the correct bucket for K. This number, *i* change as the file grows.

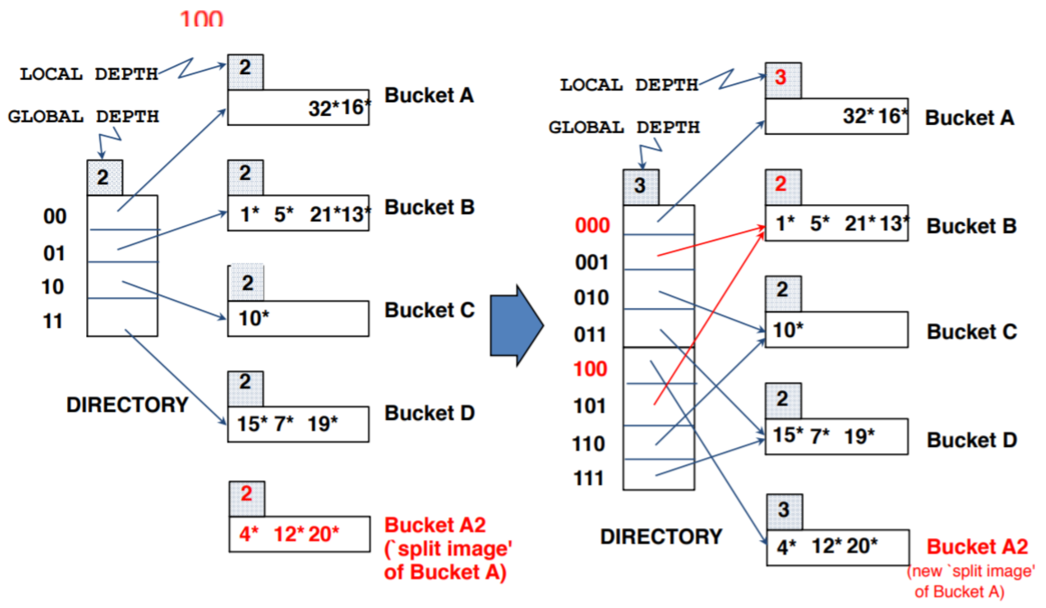
2. Queries and Updates
   - **Querying** − To locate the bucket containing search-key value K_l , the system takes the first *i* high-order bits of h(K_l), looks at the corresponding table entry for this bit string, and follows the bucket pointer in the table entry.
   - **Update** − Perform a query as above and update the data.
   - **Deletion** − Perform a query to locate the desired data and delete the same.
     - 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
   - **Insertion** − Compute the address of the bucket
     - If the bucket is empty
       - Add data to the bucket
     - Else - the bucket is full
       - The bucket must be split and insertion reattempted.
     - To split a bucket *j* when inserting record with search-key value k_j:
       - If i > i_j (more than one pointer to bucket j)
         - allocate a new bucket z, and set i_j and i_z to the old i_j -+ 1.
         - make the second half of the bucket address table entries pointing to j to point to z
         - remove and reinsert each record in bucket j.

        - recompute new bucket for K_j and insert record in the bucket (further splitting is required if the bucket is still full)
    - If $i = i_j$ (only one pointer to bucket j)
        - increment i and double the size of the bucket address table.
        - replace each entry in the table by two entries that point to the same bucket.
        - recompute new bucket address table entry for K_j Now $i > i_j$ so use the first case above.
- When inserting a value, if the bucket is full after several splits (that is, i reaches some limit b) create an overflow bucket instead of splitting bucket entry table further.

Example of Insertion - Insert 20

- $20_2 = 10100$
- Should be inserted into bucket 00, but it's already full
- To split, consider last 3 bits of 10<u>100</u>. Last two bits are the same for all entries in current bucket 00, so we use the third bit to distinguish them.





- ***Global Depth***: Max # of bits needed to tell which bucket an entry belongs to
- ***Local Depth***: # of bits used to determine if an entry belongs to this bucket

Another detailed example of insertion - p518 in the book

3. Static Hashing vs. Dynamic Hashing
    - Benefits of extendable hashing:
        - Hash performance does not degrade with growth of file
        - Minimal space overhead
    - Disadvantages of extendable hashing
        - Extra level of indirection to find desired record
        - Bucket address table may itself become very big (larger than memory)
            - Need a tree structure to locate desired record in the structure!
        - Changing size of bucket address table is an expensive operation
    - Linear hashing is an alternative mechanism which avoids these disadvantages at the possible cost of more bucket overflows

VII. Comparison of Ordered Indexing and Hashing

> Typically, ordered indexing is used unless it is known in advance that range queries will be infrequent, in which case hashing is used.
>
> Hash organizations are particularly useful for temporary files created during query processing, if lookups on a key value are required and no ranges queries will be performed.

VIII. Index Definition in SQL

- Create an index

  > **create index on** ()

  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**