Databases and Information Systems CS303

Indexing and Hashing
11-10-2023

Motivation

- Some common queries:
  - Find names of all instructors in CS department
    - Inefficient to read all instructors who are not in CS
  - Find total credits of student with ID 12345
    - Inefficient to read total credits of all other students
- Mostly for queries based on search keys
- Can we locate the relevant records directly?
  - Doable with additional storage and data structures

Indexing

- Similar to Index for a book
- Index of CS will tell us where in the disk all instructors of CS are present
- Index to Student ID 12345 will tell us where in the disk all courses / credits of that student are present
- Size of Index set is much smaller than the database itself
- Two types of index:
  - Ordered indexing Based on the ordering of values
  - Hash indexing Based on uniform distribution of values across a range of buckets

Indexing

- There are many techniques to achieve Indexing

- Factors to consider while evaluating the usefulness of a technique
  - Access types: Include finding records with a specified attribute value and finding records whose attribute values fall in a specified range.
  - Access time: The time it takes to find a particular data item, or set of items, using the technique in question.
  - Insertion time: The time it takes to insert a new data item. Includes times taken to:
    - find the correct place to insert the new data item
    - time it takes to update the index structure.
  - Deletion time: The time it takes to delete a data item. Includes times taken to:
    - Find the item to be deleted
    - Update the index structure.
  - Space overhead: The additional space occupied by an index structure.
    - If reasonable, it is usually better to sacrifice the space to achieve improved performance.

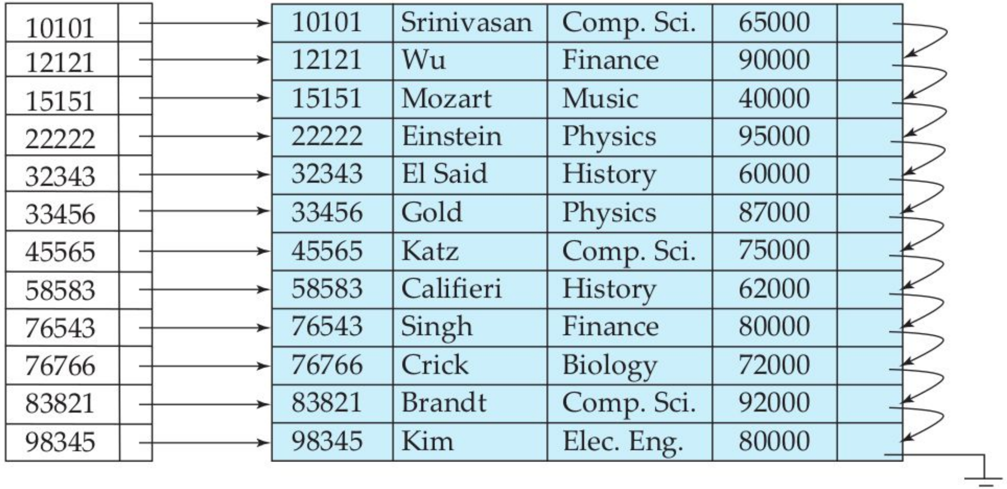
Ordered Indexing

- Indexing depends on a particular search key
  - Example : Student ID, Department etc
- Uses some ordering over the search key to organize the indices.
- Same file may have several indices on different search keys
- The file itself can be organized sequentially with respect to clustering index
  - Also called as primary indices
  - Other indices are called non-clustering indices or secondary indices

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	

Dense and Sparse Indices

- Dense Index : An index entry appears for every search-key value in the file.
- The index record contains the search-key value and a pointer to the first data record with that search-key value.
- The rest of the records with the same search-key value would be stored sequentially after the first record.
- In a dense nonclustering index, the index must store a list of pointers to all records with the same search-key value.



Dense and Sparse Indices

- Dense Index : An index entry appears for every search-key value in the file.
- The index record contains the search-key value and a pointer to the first data record with that search-key value.
- The rest of the records with the same search-key value would be stored sequentially after the first record.
- In a dense nonclustering index, the index must store a list of pointers to all records with the same search-key value.

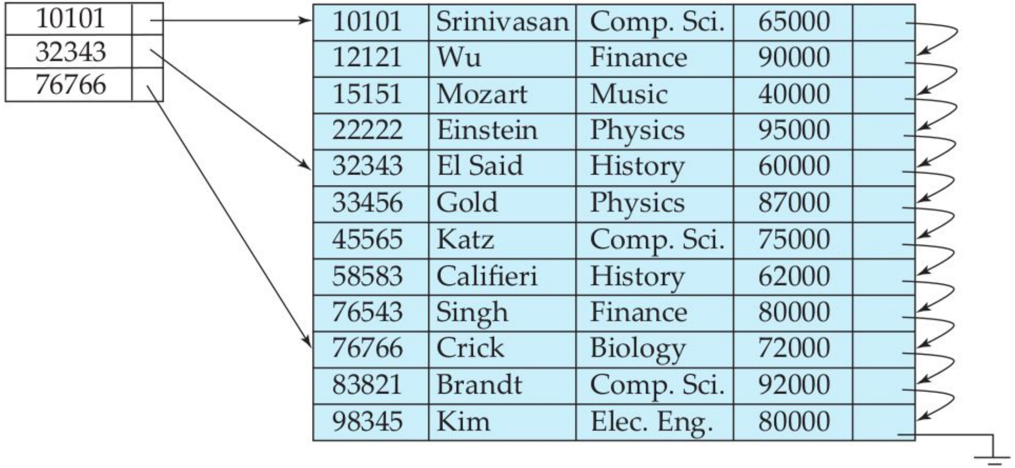
Biology	
Comp. Sci.	
Elec. Eng.	
Finance	
History	
Music	
Physics	

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 and Sparse Indices

- Sparse Index : An index entry appears for only some of the search-key values.

- Used only on the clustering index.



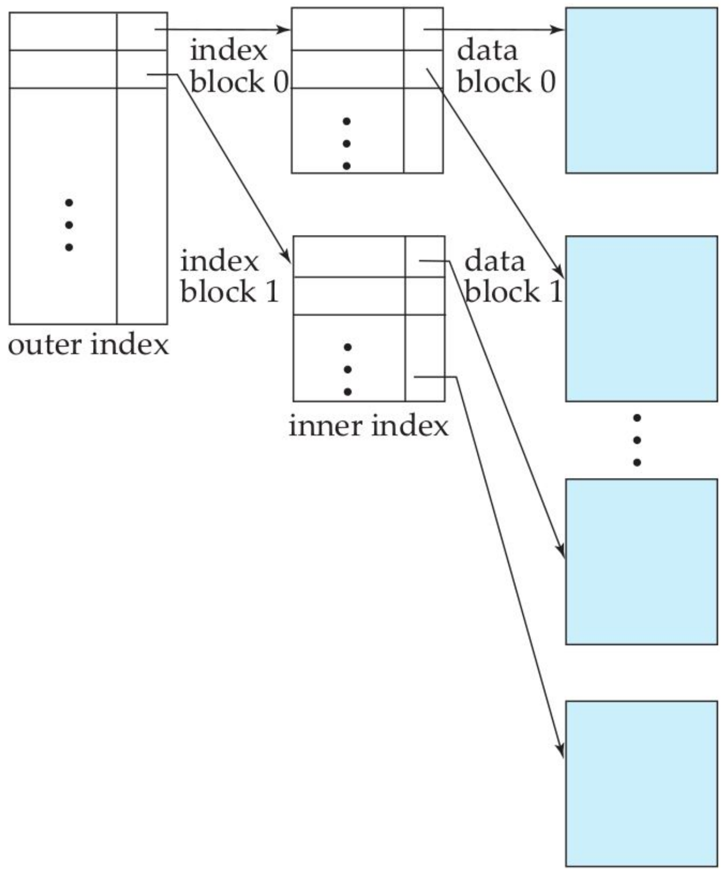
Dense and Sparse Indices

- Faster to locate a record if we have a dense index rather than a sparse index.
- But sparse indices require less space and they impose less maintenance overhead for insertions and deletions.
- In Implementation:
    - Sparse index with one index entry per block.
        - Dominant cost in processing a database request is the time that it takes to bring a block from disk into main memory.
        - Once we have brought in the block, the time to scan the entire block is negligible.

Multilevel indices

- What if index set is very large that it cannot fit in main memory?
  - Assume 10,00,000 tuples with dense index set
  - Suppose 100 entries fit in each block (of size say 4KB)
    - Then index occupies 10,000 blocks (4GB)

- Solution:
  - Use binary search to find index
    - Requires $\log_2(b)$ number of block accesses (where indices take b blocks)
      - For 10,000 blocks we need 14 block accesses
  - Treat index set itself as a smaller database
    - Outer index is sparse since inner index is always sorted
    - One index per block
    - Example needs 10000 outer index (100 blocks)
    - Can have more levels (multilevel)



Index Update

- Updating single level indices:

- Insertion of a new tuple:
  - Dense index:
    - If the search-key value does not appear in the index, the system inserts an index entry with the search-key value in the index at the appropriate position.
    - Otherwise the following actions are taken:
      - 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.
      - Otherwise, 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 records with the same search-key values.

Index Update

- Updating single level indices:

- Insertion of a new tuple:
  - Sparse index: (Assume that the index stores an entry for each block)
    - If the system creates a new block, then insert the first search-key value appearing in the new block into the index.
    - 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.

Index Update

- Updating single level indices:

- Deletion of a new tuple:
  - Dense index:
    - If the deleted record was the only record with its particular search-key value, then the system deletes the corresponding index entry from the index.

    - Otherwise the following actions are taken:
      - If the index entry stores pointers to all records with the same search-key value, the system deletes the pointer to the deleted record from the index entry.

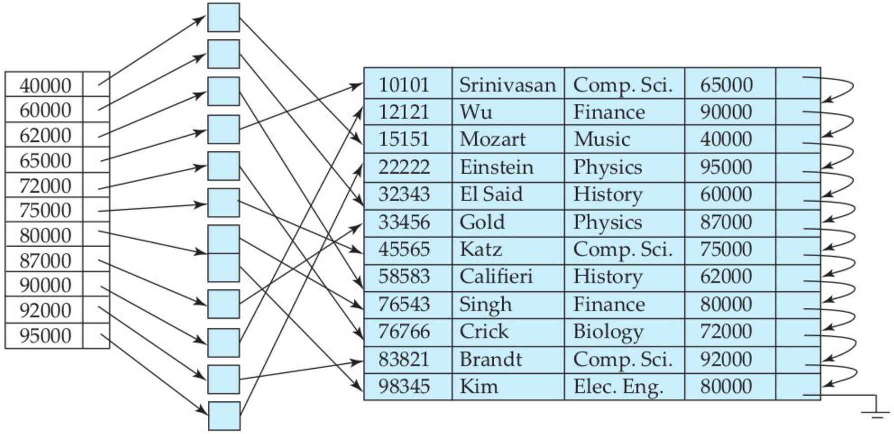
      - Otherwise, the index entry stores a pointer to only the first record with the search-key value. In this case, if the deleted record was the first record with the search-key value, the system updates the index entry to point to the next record.

Index Update

- Updating single level indices:
- Deletion of a new tuple:
    - Sparse index:
        - 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 the system takes the following actions:
            - 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.
            - Otherwise, 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.

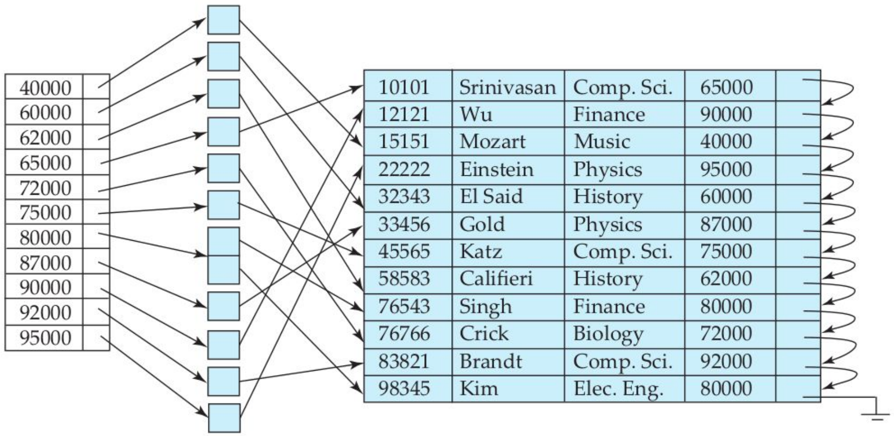
Secondary indices

- Index over the nonclustering search-keys
- Must be dense with an index for every search key and pointer to every record
- Use an extra level of indirection to implement secondary indices on search keys that are not candidate keys.
    - The pointers in such a secondary index do not point directly to the file.
    - Each points to a bucket that contains pointers to the file.



Secondary indices

- Does not ensure sequential scan of data.
- Different records with same search-key can be stored in different blocks
- Insertion / Deletion of index happens as described earlier
- Used only if there is high frequency of queries on this search-key



Indices on multiple keys

- Composite search Keys : Search-keys with more than one attributes
- The ordering of index is generally lexicographic order of the search attributes

B^+ Trees

- Disadvantage of index sets : Performance degrades as file size grows
- B^+ - tree index structure is the most widely used index structures to maintain efficiency despite insertion and deletion of data.
- B^+ - tree index takes the form of a balanced tree in which every path from the root of the tree to a leaf of the tree is of the same length.
    - Each non-leaf node in the tree has between n/2 and n children, where n is fixed for a particular tree.

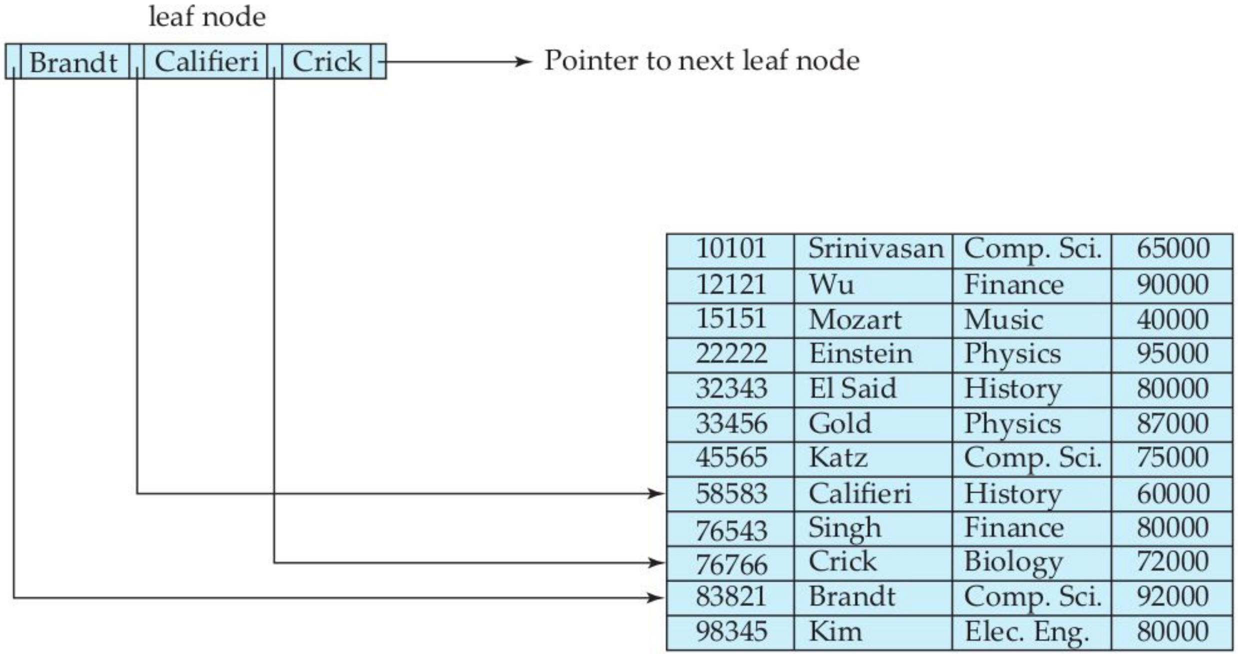
B^+ Trees : Structure

- Let n be the maximum number of children a node can have
- Each node contains upto n-1 search-keys $(K_1 \ldots K_n)$ and n Pointers $(P_1 \ldots P_n)$
- Search keys in a node are stored in an order
  - If $i < j$ then $K_i < K_j$

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

B^+ Trees : Structure

- Leaf Node :
  - For i = 1 to n-1   Each pointer P_i in leaf node points to a record in the database
  - Pointer P_n points to the next leaf node



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

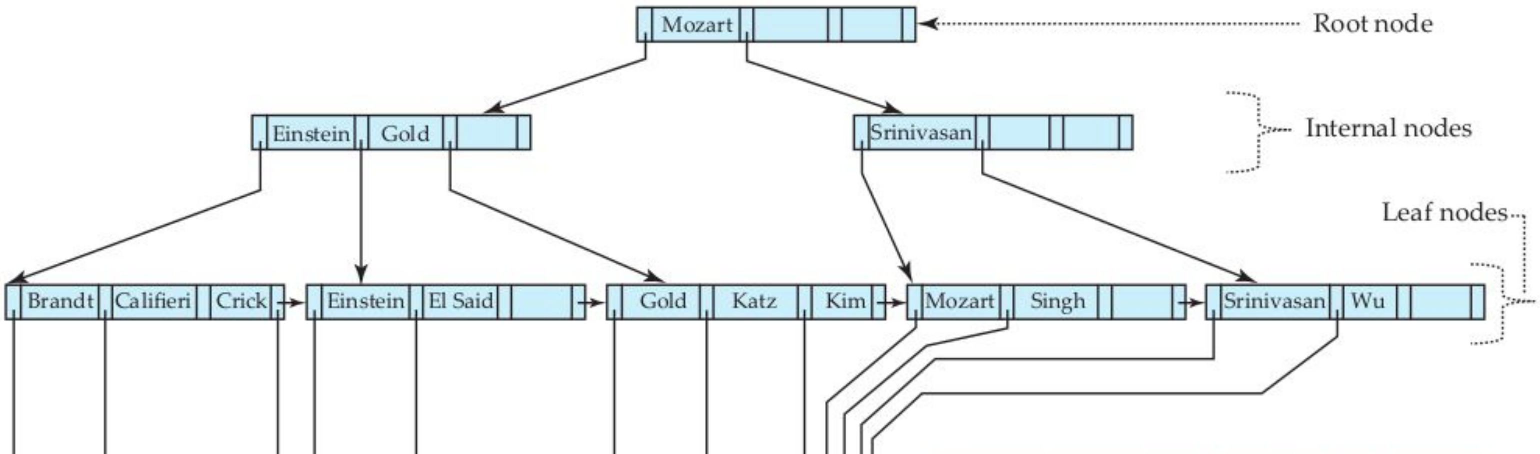
B^+ Trees : Structure

- Leaf Node :
  - For i = 1 to n-1 Each pointer P_i in leaf node points to a record in the database
  - Pointer P_n points to the next leaf node
  - Leaf node can have at most n search-keys and at least ⌈(n-1)/2⌉ search-keys
    - With k search-keys, the node has k+1 pointers
  - Values in leaf nodes do not overlap
    - Except if there are duplicates search-key values
  - B^+ - tree is a dense index (every search-key appears in some leaf node)

B^+ Trees : Structure

- Non-Leaf Node :
  - Forms multi-level index on the leaf nodes (sparse)
  - Can have at most n search-keys and at least ⌈n/2⌉ search-keys
    - With k search-keys, the node has k+1 pointers
  - Root node contains at most ⌈n/2⌉search-keys and at least 2 search-keys
  - Every pointer points to a node in the tree

B$^+$ Trees : Structure



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