BBM371- Data Management

Lecture 4: Index Files

01.11.2018

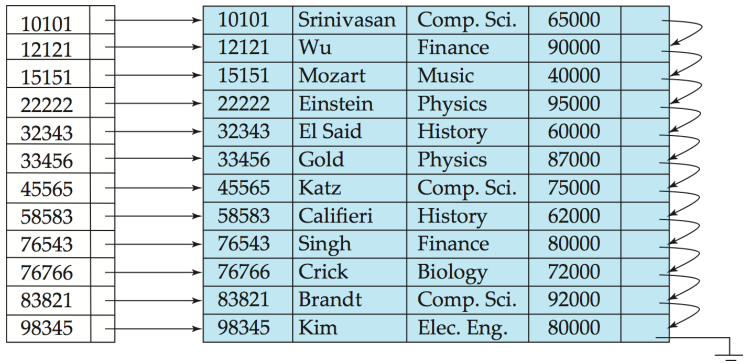
A Simple Index

- Consider the query: *SELECT * FROM R*
  We have to scan all records in R
- For *SELECT * FROM R WHERE a=10*
  It is possible to look at only a small fraction of records in R with a good indexing strategy.
- A few comments about our Index Organization:
  - The index is easier to use compared to the data file because
    1) it uses fixed-length records
    2) it is likely to be much smaller than the data file.
  - By requiring fixed-length records in the index file, we impose a limit on the size of the primary key field. This could cause problems.
  - The index could carry more information than the key and reference fields. (e.g., we could keep the length of each data file record in the index as well).

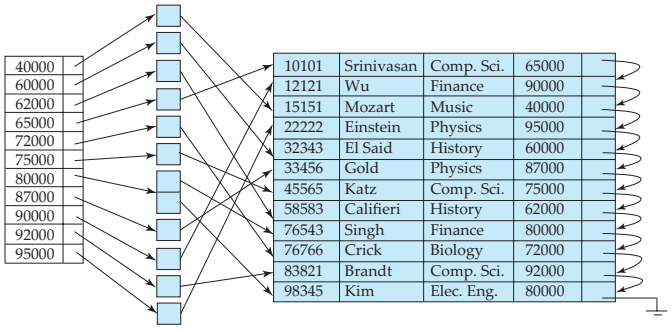
Some Terminology

- A data file stores the complete records.
- Data file can have one or more index files.
- Index files stores search keys and pointers to data-file records.
- Indexes can be primary or secondary
  - Primary Index: Data file is structured with respect to its search key. E.g. Primary key of table
  - Secondary Index: Built on some other attributes of the table, the order of records in the data-file are independent from the search key
- Sequential file: Records are sorted with respect to the primary key.

Primary Index Example

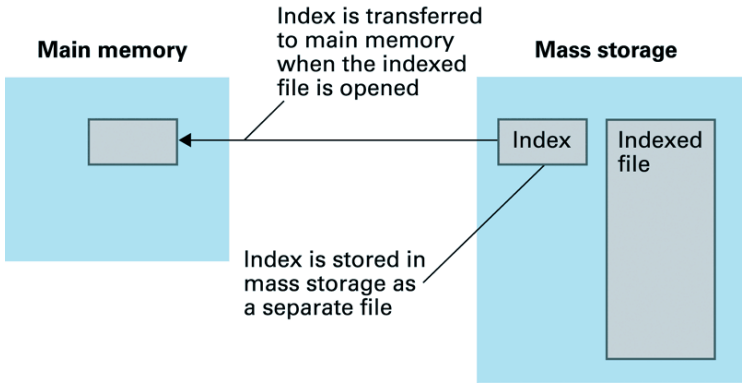


Secondary Index Example



Secondary index on *salary* field of *instructor*

Opening an indexed file



Basic Operations on an Indexed Entry-Sequenced File

- Assumption: the index is small enough to be held in memory. Later on, we will see what can be done when this is not the case.
  - Create the original empty index and data files
  - Load the index into memory before using it.
  - Rewrite the index file from memory after using it.
  - Add records to the data file and index.
  - Delete records from the data file (and index file).
  - Update records in the data file if update changes the key update the index file.

Creating, Loading and Re-writing

- The index is represented as an array of records. The loading into memory can be done sequentially, reading a large number of index records (which are short) at once.
- What happens if the index changed but its re-writing does not take place or takes place incompletely?
  - Use a mechanism for indicating whether or not the index is out of date.
  - Have a procedure that reconstructs the index from the data file in case it is out of date.

Record Addition

- When we add a record, both the data file and the index should be updated.
- In the data file the **byte-offset** of the new record should be saved.
  - If it is a simple **heap file**, the record can be added to an arbitrary location.
  - If it is a **sequential file**, the record should be added with respect to its primary key
- Since the index is sorted, the location of the new search key does matter: we have to shift all the keys that belong after the one we are inserting to open up space for the new record. However, this operation is not too costly as it is performed in memory.

Record Deletion

- Record deletion can be done using several methods
- In addition, however, the index record corresponding to the data record being deleted must also be deleted. Once again, since this deletion takes place in memory, the record shifting is not too costly.

Record Updating

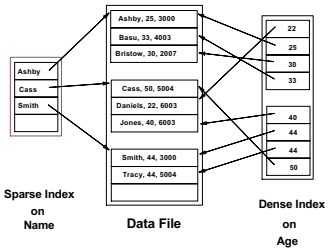
- Record updating falls into two categories:
  - The update changes the value of the index key field.
  - The update does not affect the index key field.
- In the first case, both the index and data file may need to be reordered. The update is easiest to deal with if it is conceptualized as a delete followed by an insert (but the user needs not know about this).
- In the second case, the index does not need reordering, but the data file may. If the updated record is smaller than the original one, it can be re-written at the same location. If, however, it is larger, then a new spot has to be found for it. Again the delete/insert solution can be used.

Indexes that are too large to hold in memory

- Problems:
  - Binary searching requires several seeks rather than being performed at memory speed.
  - Index rearrangement requires shifting or sorting records on secondary storage ==> Extremely time consuming.
- Solutions:
  - Use a hashed organization
  - Use a tree-structured index (e.g., a B-Tree)

Index Classification

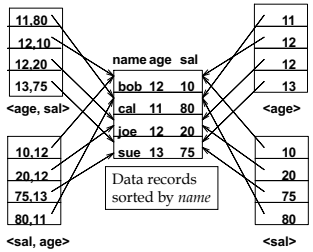
- Dense vs. Sparse: If there is at least one data entry per search key value (in some data record), then dense.
- Sparse indexes points only to a group of records, perhaps to first records in a block
- Sparse indexes are smaller; however, some useful optimizations are based on dense indexes.

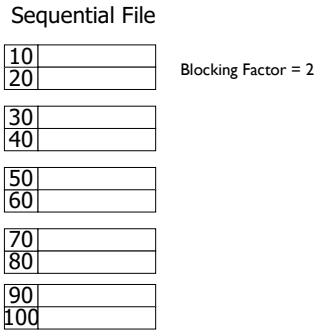


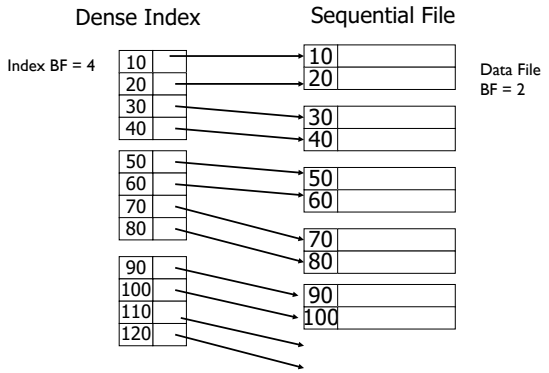
Composite Search Keys

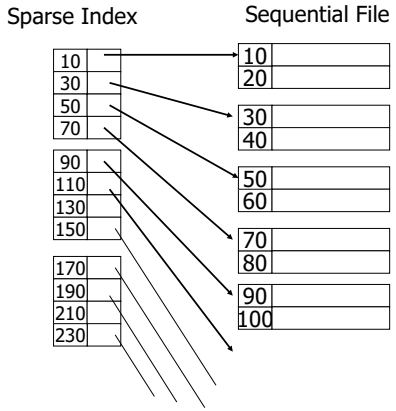
- Composite Search Keys: Search on a combination of fields.
  - Equality query: Every field value is equal to a constant value. E.g. <sal,age> index:
    - age=20 and sal =75
  - Range query: Some field value is not a constant. E.g.:
    - age=20 and sal > 10
- Using index <age, sal> :
  - Can find age=20 and sal>10 efficiently
  - Not efficient for age>20 and sal=10

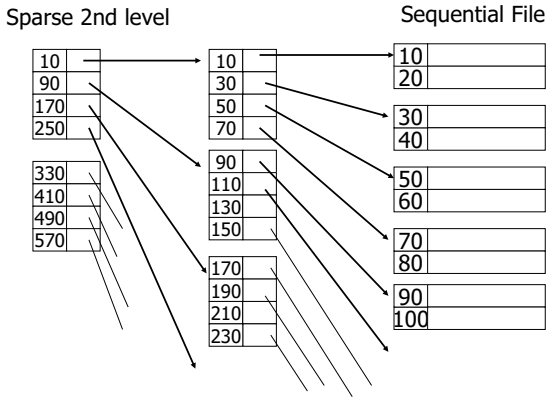
Examples of composite key indexes using lexicographic order.







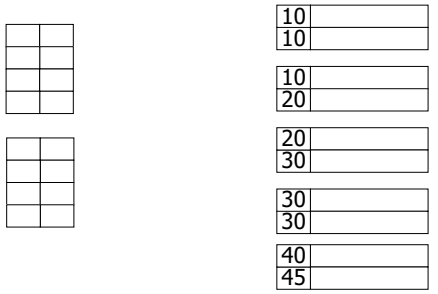




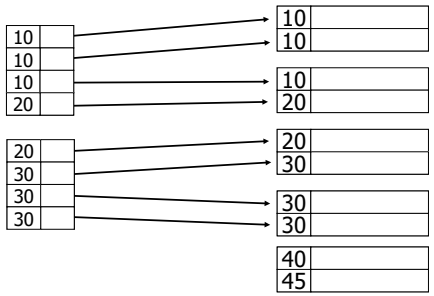
Sparse vs. Dense Tradeoff

- Sparse: Less index space per record can keep more of index in memory
  - The data-file must be sorted w.r.t. search key.
- Dense: Can tell if any record exists without accessing file
  - Can work even if the data file is not ordered!

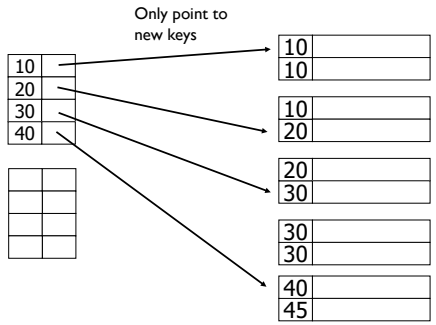
Duplicate keys



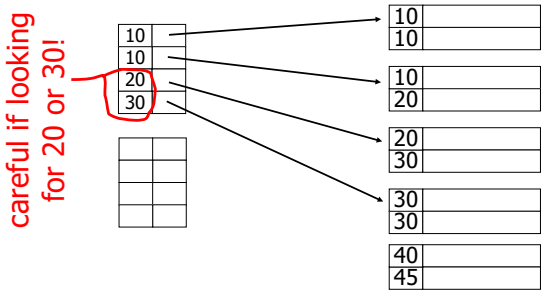
Dense index, one way to implement?



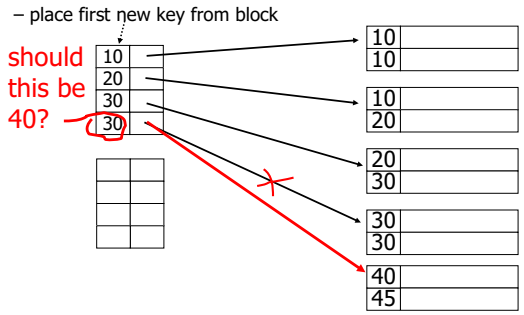
Duplicate Keys (Alternative)



Sparse index, one way?



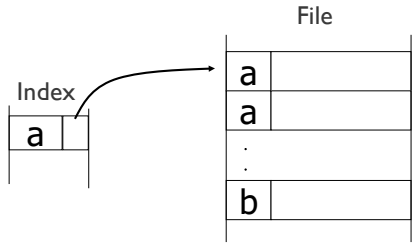
Sparse index, another way?



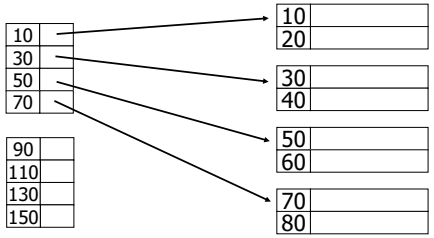
Duplicate values, primary index

Summary

- Index may point to first instance of each value only

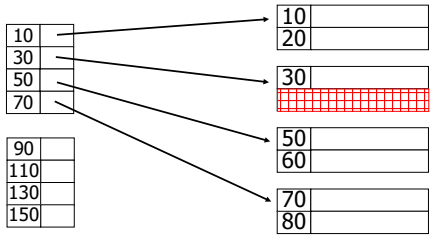


Deletion from sparse index



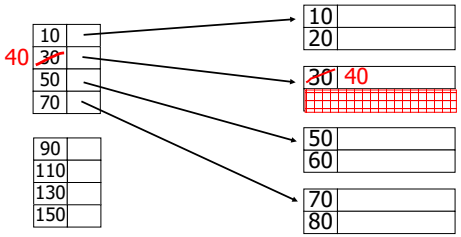
Deletion from sparse index

– delete record 40



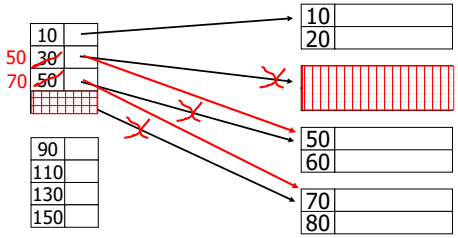
Deletion from sparse index

– delete record 30

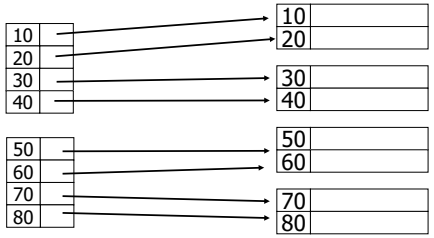


Deletion from sparse index

– delete records 30 & 40

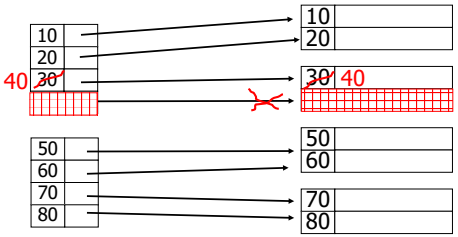


Deletion from dense index

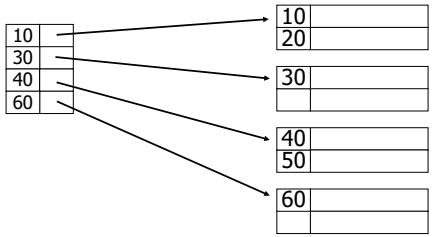


Deletion from dense index

– delete record 30

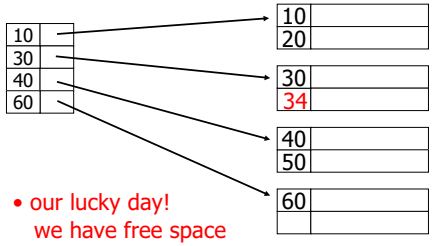


Insertion, sparse index case



Insertion, sparse index case

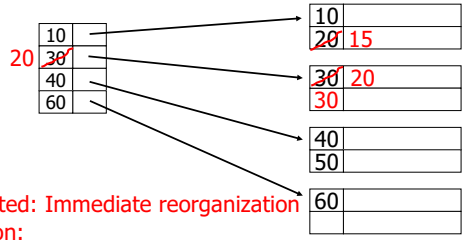
– insert record 34



- our lucky day! we have free space where we need it!

Insertion, sparse index case

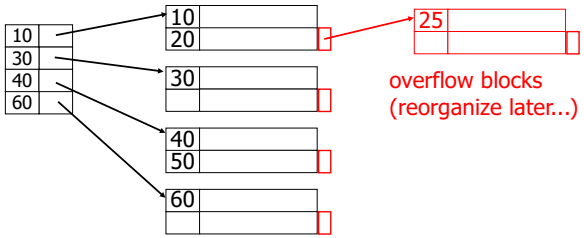
– insert record 15



- Illustrated: Immediate reorganization
- Variation:
  - insert new block (chained file)
  - update index

Insertion, sparse index case (alternate)

– insert record 25

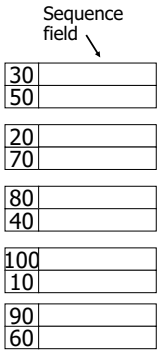


overflow blocks (reorganize later...)

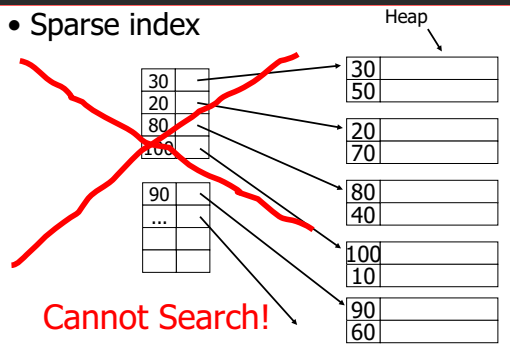
Insertion, dense index case

- Similar
- Often more expensive . . .

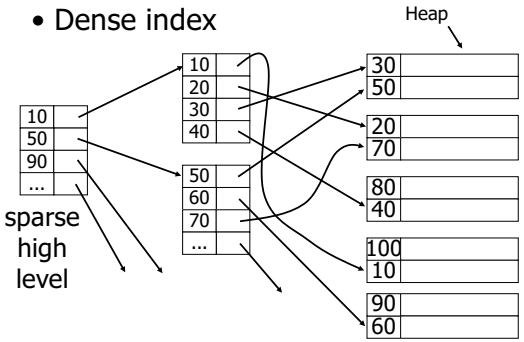
Continues



Sparse Index only for Sequential Data-file



Multilevel indexes



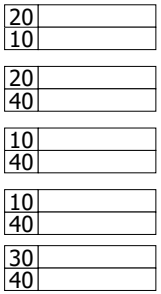
With multilevel indexes:

- Sparse index; Lower-level should be sorted
  - Data-file is sorted, we can use sparse index
  - Sparse index on top of sparse even if data-file is Unordered!
- Other levels are sparse
  - Why not Dense??

Also: Pointers are record pointers

(not block pointers; not computed)

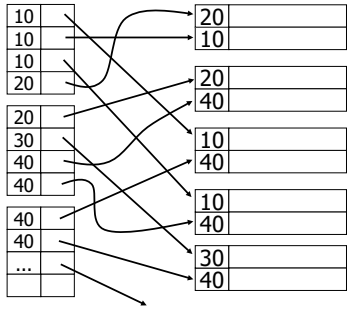
Duplicate values & secondary indexes



Duplicate values & secondary indexes

one option...

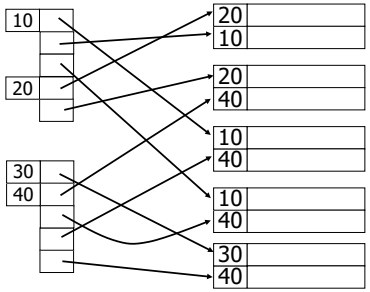
Problem:
excess overhead!
- disk space
- search time



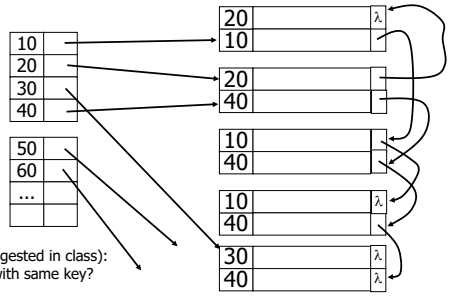
Duplicate values & secondary indexes

another option...

Problem:
variable size records in index!



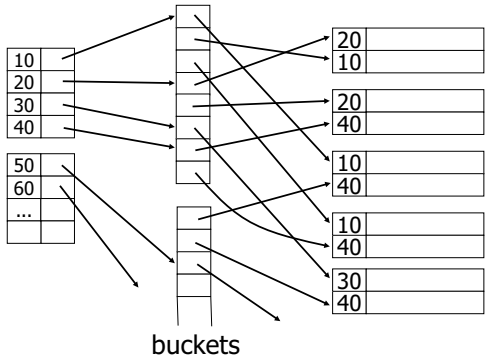
Duplicate values & secondary indexes



Another idea (suggested in class):
Chain records with same key?

Problems:
- Need to add fields to records
- Need to follow chain to know records

Duplicate values & secondary indexes



Example (Question)

- Suppose there is a data file of 4 GB (2^{32}) in a system with blocks of 1KB and fixed length records of 256Bytes.
- The records are stored in sorted order with respect to the key Student ID.
- Index stores a search key of 4 Bytes and a 4 Bytes of pointer. So, an index entry is 8 bytes.
- How many disk accesses do we need to find a record with a given Student ID:
  - Using sorted data file
  - Using dense index
  - Using sparse index

Example

- 2^{32} / 2^{10} = 2^{22} blocks are in the data file.
- Blocking Factor of data file = $2^{10}/2^{8} = 2^{2}$
- 2^{22} x BF of data file = 2^{24} records in the file
- With binary search we can have 22 disk accesses to find the record we are searching for in the worst case.
- If an index entry is 8 bytes we can fit into a block $2^{10}/2^{3}=2^{7}$ entries
  - Dense Index should be: $2^{24}/ 2^{7}=2^{17}$ blocks = 2^{17} x 2^{10} = 2^{27} = 128MB index file. So, a binary search is 17 disk accesses.
  - Sparse Index should be: $2^{22}/2^{7}=2^{15}$ blocks = 2^{15} x 2^{10} = 2^{25} = 32MB index file. So, a search is 15 disk accesses.
- What would happen if we had a two-level sparse index?