CS525: Advanced Database Organization

Notes 5: Indexing and Hashing
Part I: Conventional indexes

Yousef M. Elmehdwi

Department of Computer Science

Illinois Institute of Technology

yelmehdwi@iit.edu

September 20^{th}, 2023

Slides: adapted from courses taught by Hector Garcia-Molina, Stanford, Elke A. Rundensteiner, Worcester Polytechnic Institute, Shun Yan Cheung, Emory University, Marc H. Scholl, University of Konstanz, Principles of Database Management ,Ellen Munthe-Kaas, Universitetet i Oslo, & Andy Pavlo, Carnegie Mellon University

Course Status

- We are now going to talk about how to support the DBMS's execution engine to read/write data from pages.
- Two types of data structures:
  - Trees:
  - Hash Tables
- *Data structures are a fundamental component of a DBMS, and they are employed for various purposes, including organizing data, optimizing query processing, and managing system metadata. These data structures play a crucial role in the efficient functioning of a database system.*

Applications of Data Structures in DBMS

- A DBMS uses various data structures for many different parts of the system internals. Some examples include:
  - Internal Meta-Data: Data structures keep track of information about the database and the system state.
  - i.e., used to manage internal meta-data, including page tables and directories.
    - Hash tables can map page IDs to frames or to locations on disk.
  - Core Data Storage: Can be used as the base storage for tuples in the database.
    - e.g., use hash table/tree to organize underlying pages/tuples inside pages
  - Temporary Data Structures: During query processing, DBMS may create temporary data structures, such as hash tables, to optimize query execution (e.g., hash tables for join operations).
  - Table Indexes: Auxiliary data structures are used to enhance the efficiency of searching for specific tuples within the database tables. Indexes provide quicker access to data based on specific criteria.

Design Decisions

- There are two main design decisions to consider when implementing data structures for the DBMS:
- Data Organization
  - In the design of data structures for a DBMS, one critical decision is determining how to organize data in memory efficiently
  - We need to figure out how to layout memory and what information to store inside the data structure in order to support efficient access.
- Concurrency
  - Another key design consideration is ensuring concurrent access to the data structure.
  - DBMSs often need to support multiple threads or users accessing data simultaneously.
  - We need to think about how to enable multiple threads to access the data structure at the same time without causing problems.

This Chapter

- How to find a record quickly, given a key

Today's Agenda

- Conventional indexes
  - Basic Ideas: sparse, dense, multi-level ...
  - Duplicate Keys
  - Deletion/Insertion
  - Secondary indexes
- `B`$^+$`-Trees`. (Next)
  - `B`$^+$`Tree` Overview
  - Design Decisions
  - Optimizations
- Hash Tables

How to Store a Table

- Suppose we scatter its records arbitrarily among the blocks of the disk
- How to answer

```
SELECT * FROM R;
```

1. We would have to examine every block in the storage system (Scan every block)
   - slow, overhead
2. Reserve some blocks for the given relation
   - Slightly better organization, no need to scan the entire disk

- How about: "*find a tuple given the value of its primary key*"

```
SELECT * FROM R WHERE condition;
```

  - Scan all the records in the reserved blocks
    - Still slow

- We need a mechanism to speed up the search for a tuple with a particular value of an attribute

Table Indexes

- A table index is a replica of a subset of a table's columns that are organized and/or sorted for efficient access using a subset of those columns.
  - i.e., a data structure that enable the user to find (locate) data items efficiently (quickly) using search keys
- The DBMS ensures that the contents of `the table` and the `indices` are in sync.
- *Indexes are data structures used to optimize the retrieval of data from a database.*

Table Indexes

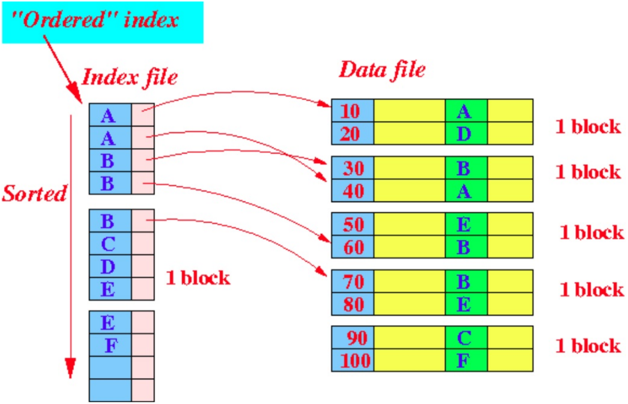
- It is the DBMS's job to figure out the best index(es) to use to execute each query.
- There is a trade-off on the number of indexes to create per database.
    - *Although more indexes make looking up queries faster, indexes also use storage and require maintenance*
    - Storage Overhead (indexes use storage)
    - Maintenance Overhead (require maintenance)

Index

- `Index`: a data structure that allows users to locate data items efficiently using search keys.
- Search Key: attribute/set of attributes used to look up records in a file.
- An index file is a file containing **key-pointer** pairs of the form `(K,a)`
  - i.e., consists of records (called index entries)
  - `K` is a **search key**, `a` is an **address/pointer to a block/record**
- Index files are typically much smaller than the original file
- The pointer is usually a block pointer
  - So the index allow you to locate the block that contains the record quickly
  - The record is found by a search operation inside the block (after the block is read into main memory)
- 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.

Ordered Index

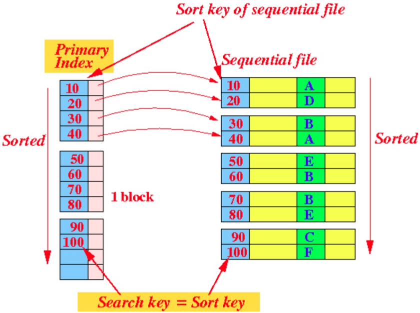
- `Ordered index`: index entries are stored sorted on the search key value



- The values in the index are ordered so that a binary search can be done.

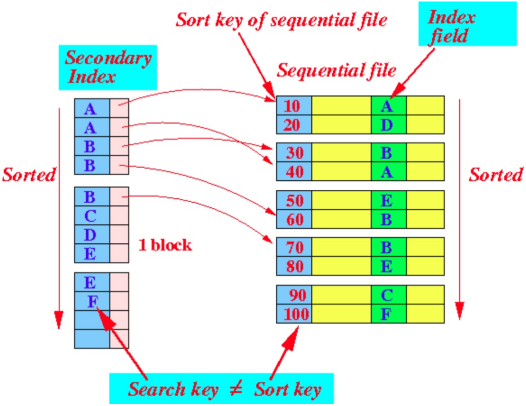
Clustering index

- A `clustered index` defines the physical storage order of data in a table, allowing for sorting in only one specific way. It is often referred to as a primary index.
- In a sequentially ordered file, the clustering index is the one where the search key specifies the sequential order of the file.
  - *This means that records are sorted based on the primary key or another designated key, which serves as the sort key.*
- The `sort key` consists of field(s) whose values determine the order in which records are arranged in a sequential file



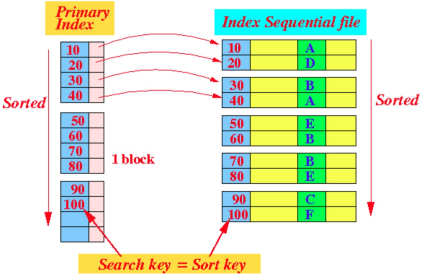
Secondary index

- `Secondary index`: an index in which the search key specifies an order that differs from the sequential order of the file.
- Also called nonclustering index.
  - *This means the secondary index allows data retrieval in a specific order other than the physical storage order of the file, providing alternative access paths to the data.*



Indexed Sequential File

- Assume all files are ordered sequentially on some search key.
- `Index-sequential file`: A type of sequential file that is ordered based on a search key. It typically includes a clustering index on the search key, which determines the physical storage order of the data.



- These files represent one of the oldest index structure schemes used in database systems.
- Designed for applications that require both sequential processing of the entire file and random access to individual records.

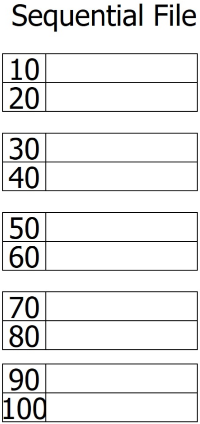
Ordered Indices: Dense and Sparse Indices

- Indexes can take various forms, including sparse indexes, dense indexes, and multi-level indexes.
  - **Dense index**: Contains entries for every data record, enabling direct access to any record.
    - one per record in sequential file
  - **Sparse index**: Contains entries only for some of the data records, typically resulting in smaller index size.
    - for every data block, i.e., the key of the first record
- Dense indexes are generally faster, but require more storage space and are more complex to maintain than sparse indexes
- Note: index file occupies fewer disk blocks than data file and can be searched much quicker

Dense Index

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

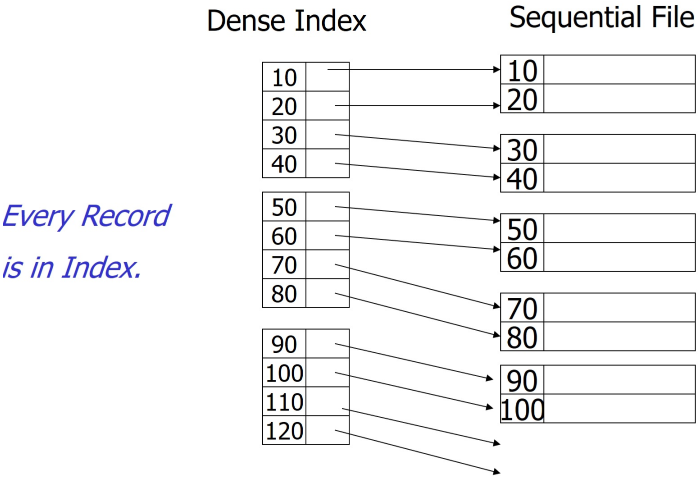
- An index with one entry for every `key` in the data file



Dense Index

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

- An index with one entry for every `key` in the data file



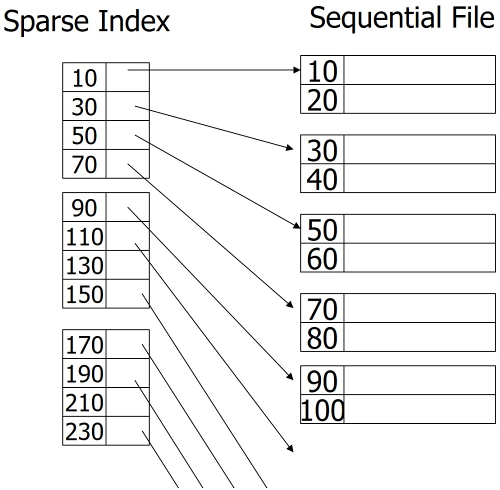
Dense Index: Advantages

- There are several factors that make dense indexes more efficient than it seems:
  - Number of index blocks usually small compared with the number of data blocks,
    - If the index is too large, use sparse index instead.
  - An existence search of a record may be done via index only.
  - The record is directly found in a block using the pointer, i.e., no search within the block.
  - If index fits in memory, a record can be found using a given search key with a maximum one disk I/O.

Sparse Index

Sparse Index: contains index records for only some search-key values.

- There is just one `(key,pointer)` pair per data block.
- The `key` is for the first record in the block.



Only first Record

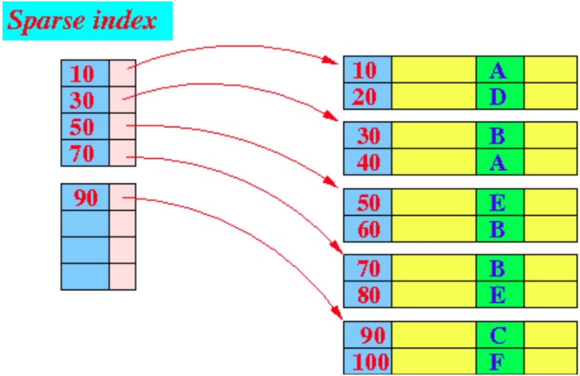
per block in Index.

Sparse Index

- only one index field per block, i.e., less index data
- cannot find out if a record exists only using the index
- to locate a record with `key K`:
  1. search the index for the largest key less than or equal to `K`
  2. retrieve the block pointed to by the index field
  3. search within the block for the record with `key K`

Using a Sparse Index

- Look up the record with `search key = 40`

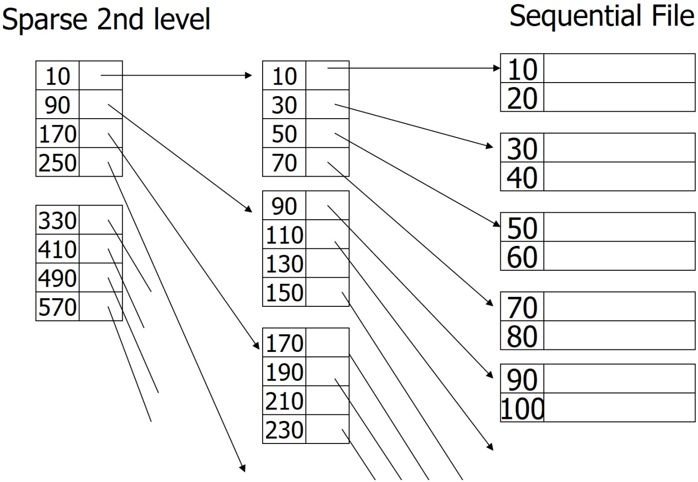


- Procedure
  - Find the largest search key that is ≤ 40. Found key $= 30$
  - Search in the data block for search key 40

Multilevel Index

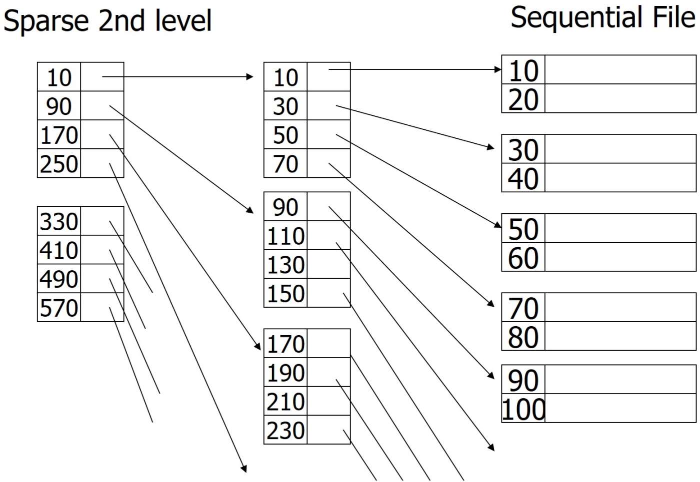
- An index itself can span many blocks, i.e., still many block accesses
- Using a multi-level index is one approach to increase efficiency, i.e., using an index on the index
    - *Utilizes multiple levels of indexing, allowing for efficient access to a large dataset.*
- If index does not fit in memory, access becomes expensive.
- Solution: treat index kept on disk as a sequential file and construct a sparse index on it.
    - **outer index** - a sparse index of the basic index
    - **inner index** - the basic 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.

Two-Level Index Example



Question

- Can we (do we want to) build a dense, 2^{nd} level index for a dense index?



Question

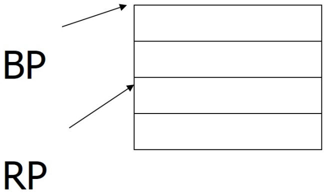
- Can we (do we want to) build a dense 2^{nd} level index for a dense index?
- YES, but it does not make sense
- Second and higher level indexes must be sparse, otherwise no savings
- The reason is that a dense index on an index would have exactly as many key-pointer pairs as the first-level index, and therefore would take exactly as much space as the first-level index.

Question

- Does it make sense to use a sparse index on an unsorted file?
- NO, how can one find records that are not in the index
- BUT, one might use a sparse index on a dense index on an unsorted file

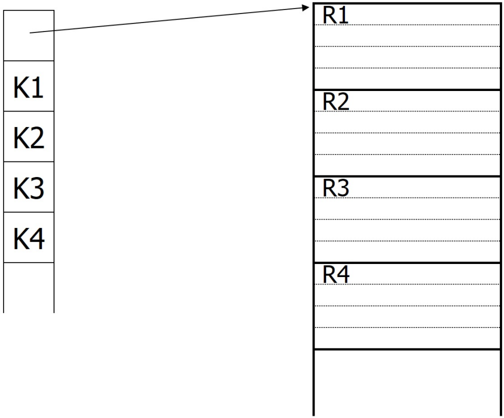
Notes on pointers

1) Block pointer (used in sparse index) can be smaller than record pointer (used in dense index)

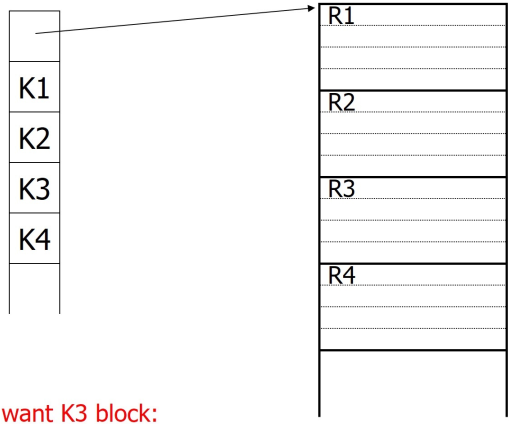


2) If file is contiguous, then we can omit pointers (i.e., compute them)

Notes on pointers



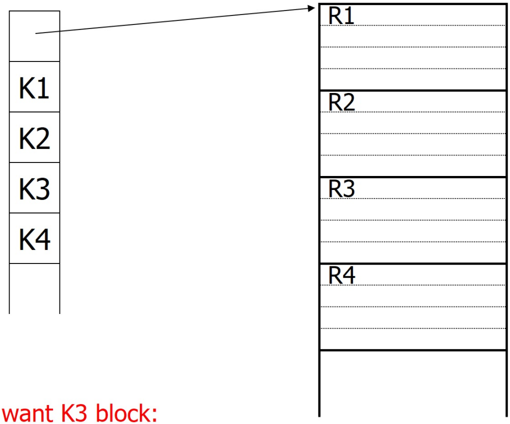
Notes on pointers



say:
1024 B
per block

- if we want K3 block:

Notes on pointers



say:
1024 B
per block

- if we want K3 block:
- get it at offset (3-1)*1024 = 2048 Bytes

Next

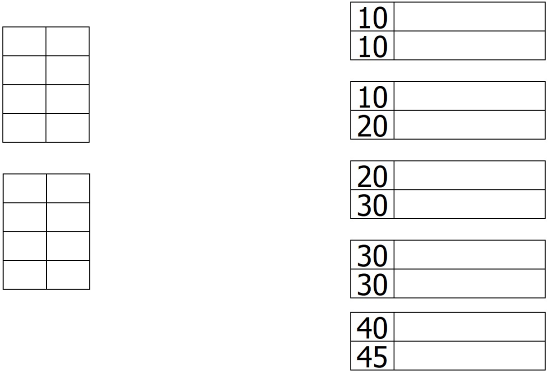
- Duplicate keys
- Deletion/Insertion
- Secondary indexes

Duplicate keys

- So far we have looked at indexes where the search key has been unique - if records also are sorted, the index is called a primary index
  - *A primary index is an index associated with a sorted file where the search key is unique.*
- In some cases, databases allow for duplicate values in search keys, meaning multiple records can have the same search key value.
- This situation requires the use of indexes that can handle duplicates efficiently.
  - Indexes are also used on non-key attributes where duplicate values are allowed
- In general, if the records are sorted by the search key, the previous ideas may be applied
- Many ways to implement such an index:
  - dense index with one index field
    - per record (pointer to all duplicates)
    - unique search key (pointer to only the first record)
  - sparse index

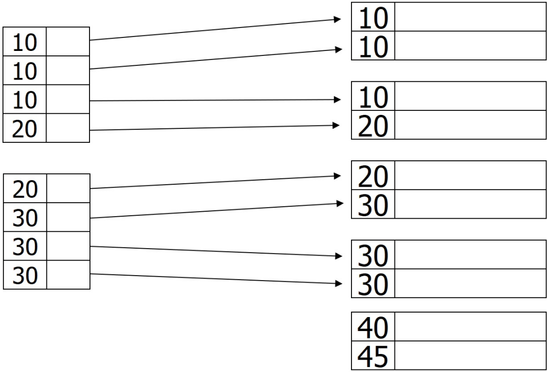
Duplicate keys

- What if more than one record has a given search key value?
- Then the search key is not a key of the relation



Duplicate Search Keys with Dense Index

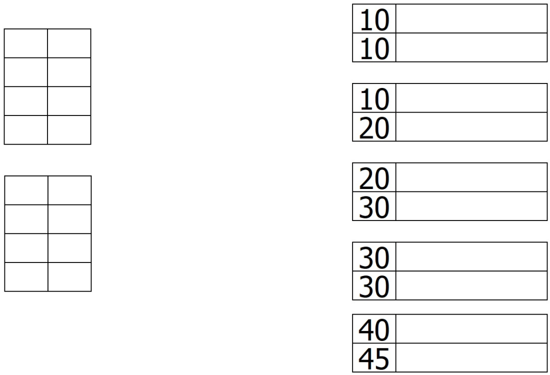
- Dense index, one way to implement? (Point to each value)
  - one entry with key `K` for each record of the data file that has search key `K`



- ☺ easy to find records and how many
- ☹ more fields than necessary? - index itself spans more disk blocks
- To find all data records with search key `K`, follow all the pointers in the index with search key `K`

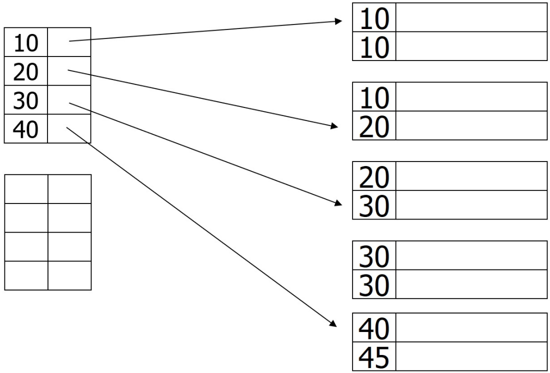
Duplicate Search Keys with Dense Index

- Dense index, better way?



Duplicate Search Keys with Dense Index keys

- Dense index, better way? Point to each distinct value!
  - only one index field per unique search key (saves some space in the index)



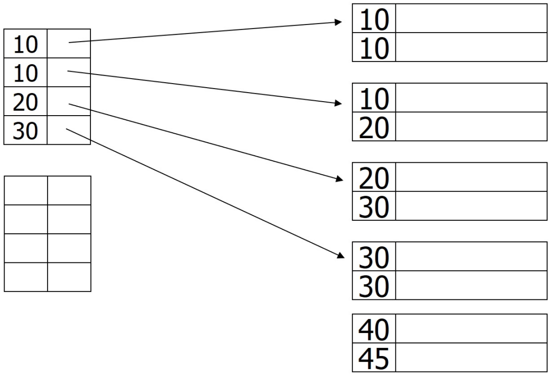
☺ smaller index - quick search

☹ more complicated to find successive records

- To find all data records with search key `K`, follow the one pointer in the index and then move forward in the data file

Duplicate search keys with sparse index

- Sparse index, one way?
  - index field is first record in each block, pointer to block



☺ small index fast search

☹ complicated to find records

  - e.g., must be careful if looking for 30

Next

- Duplicate keys
- Deletion/Insertion
- Secondary indexes

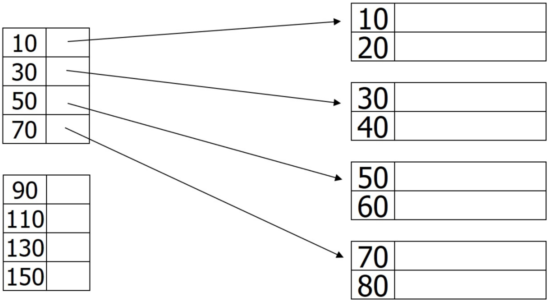
Modifications

- An index file is a sequential file and must be treated in a similar way as a file of sorted records:
  - use overflow blocks
  - insert new blocks
  - slide elements to adjacent blocks
- A **dense index** points to the records, i.e.:
  - modified if a record is created, deleted, or moved
  - no actions must be done on block operations
- A **sparse index** points to the blocks, i.e.:
  - may be modified if a record is created, deleted or moved
  - no action must be done managing overflow blocks (pointers to primary blocks only)
  - must insert (delete) pointer to new (deleted) sequential block

Deletion using Sparse Index

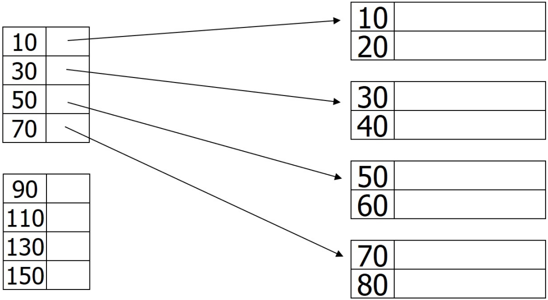
- If an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order).
- If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

Deletion using Sparse Index



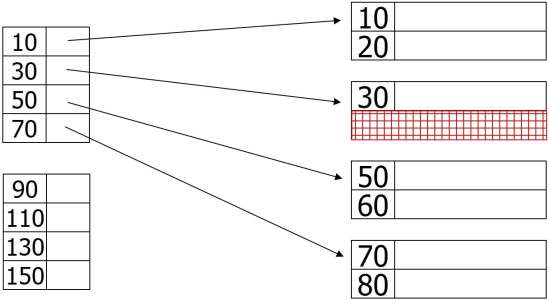
Deletion using Sparse Index

delete record 40



Deletion using Sparse Index

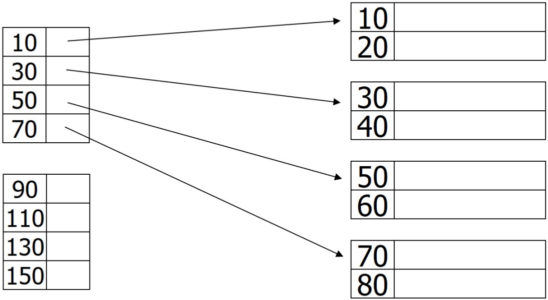
delete record 40



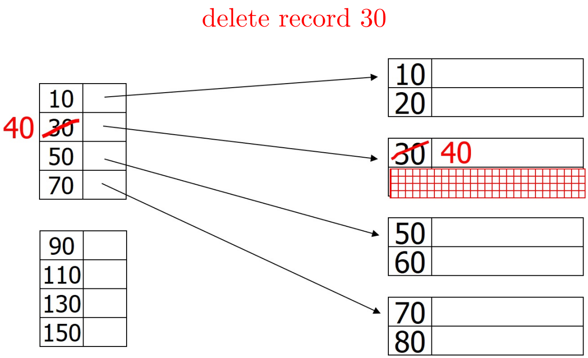
- as a sparse index points to the block, no action is required

Deletion using Sparse Index

delete record 30



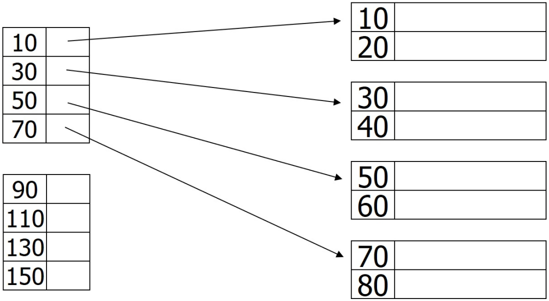
Deletion using Sparse Index



- the first record of the block has been updated, i.e., the index must also be updated
- Assumption: no pointers to records from anywhere

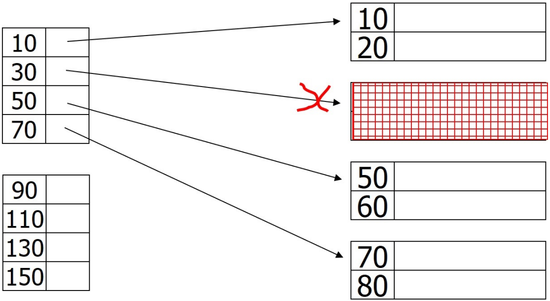
Deletion using Sparse Index

delete records 30 & 40

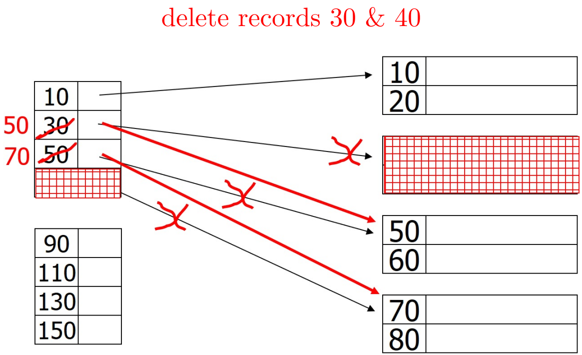


Deletion using Sparse Index

delete records 30 & 40



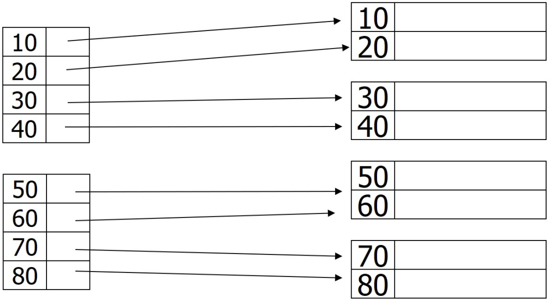
Deletion using Sparse Index



- Since the second data block no longer exists, we delete its entry from the index
- Optional: the first index block being consolidated by moving forward the following pairs

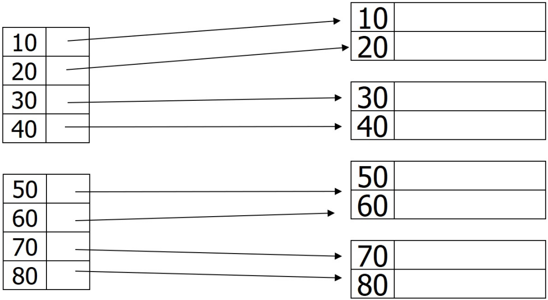
Deletion using Dense Index

- Deletion of search-key is similar to file record deletion.



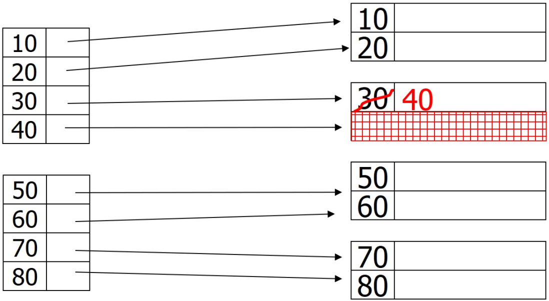
Deletion using Dense Index

delete record 30



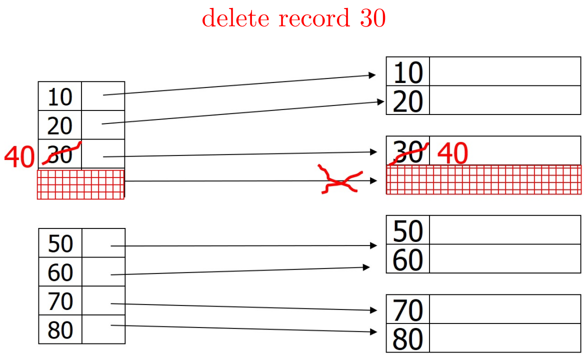
Deletion using Dense Index

delete record 30



- Assumption: no pointers to records from anywhere

Deletion using Dense Index

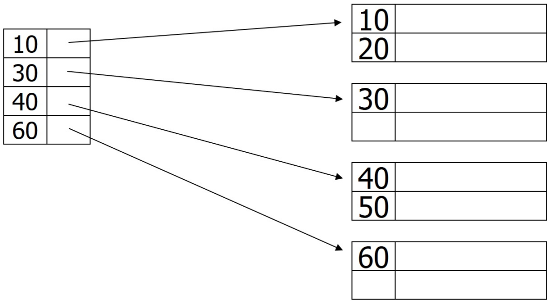


- in many cases it is convenient to “compress” data in the blocks (optional)
- one might compress the whole data set, but one usually keep some free space for future evolution of the data

Insertion, sparse index

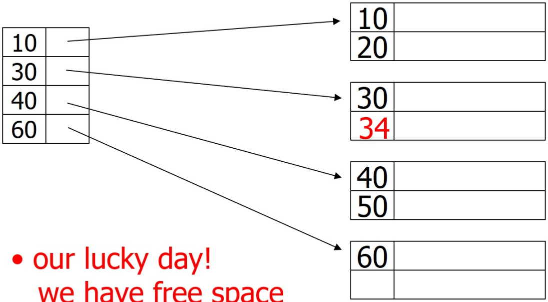
- Perform a lookup using the search-key value of the record to be inserted.
- If index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created.
  - If a new block is created, the first search-key value appearing in the new block is inserted into the index.

Insertion, sparse index case



Insertion, sparse index case

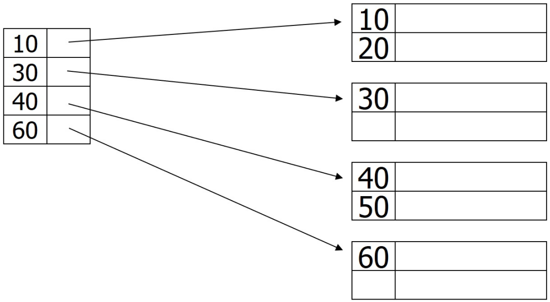
insert record 34



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

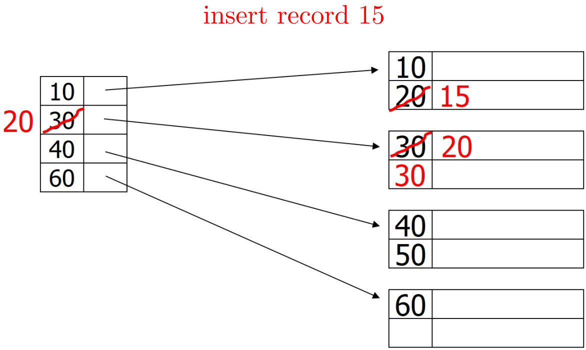
Insertion, sparse index case

insert record 15



- One Way: Look for a nearby block with some extra space, thus slide blocks backward in the file to make room for record `15`.

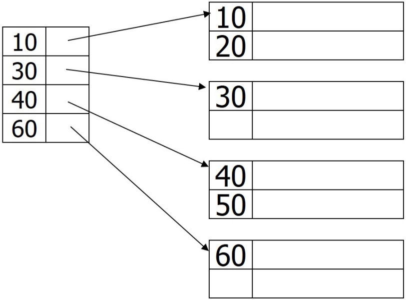
Insertion, sparse index case



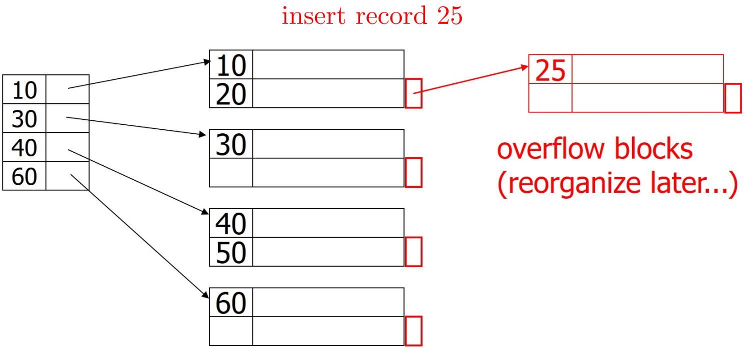
- Illustrated: Immediate reorganization
  - record 15 should go into first block, has to move record 20 to second block where we have room
  - first record of block 2 has changes, must also update index
- Alternative: instead of sliding record 20, we might have inserted a new block or an overflow block

Insertion, sparse index case

insert record 25



Insertion, sparse index case



- no available room - insert overflow block or new sequential block
- overflow block
  - no actions are required in the index, sparse indexes only have pointers to primary blocks
- new sequential (primary) block
  - must update index

Insertion, dense index case

- Similar, but must be updated each time
- Often more expensive

Sparse vs. Dense Tradeoff

	Dense	**Sparse**
space	one index field per record	one index field per data block
block accesses	many	few
record access	direct access	must search within block
exist queries	use index only	must always access block
use	anywhere (not dense-dense)	not on unordered elements
modification	always updated if the order of records change	updated only if first record in block is changed

Topics

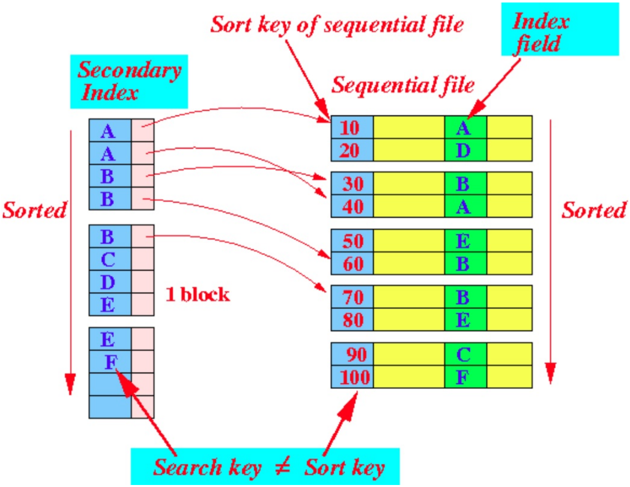
- Conventional indexes
  - Basic Ideas: sparse, dense, multi-level ...
  - Duplicate Keys
  - Deletion/Insertion
  - Secondary indexes
- B^+-Trees
- Hashing schemes

Secondary Indexes

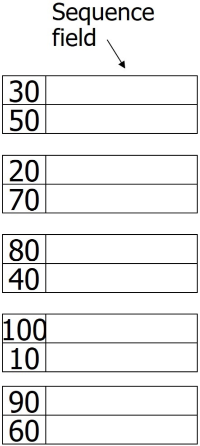
- Secondary indexes are additional indexes created on attributes other than the primary key.
- They provide alternative access paths to the data, allowing for efficient retrieval based on non-primary key attributes.
- Sometimes we want multiple indexes on a relation.
  - Ex: search `Candies(name,manf)` both by `name` and by `manufacturer`
- Typically the file would be sorted using the key (ex: `name`) and the primary index would be on that field.
- The secondary index is on any other attribute (ex: `manf`).
- Secondary indexes
  - works on unsorted records
  - works like any other index - find a record fast
  - first level is always dense - any higher levels are sparse
  - duplicates are allowed
  - index itself is sorted on the search key value - easy to search
- *Secondary indexes serve the same purpose as primary indexes, but search key specifies an order different from the sequential order of the file*

Recall: Secondary index

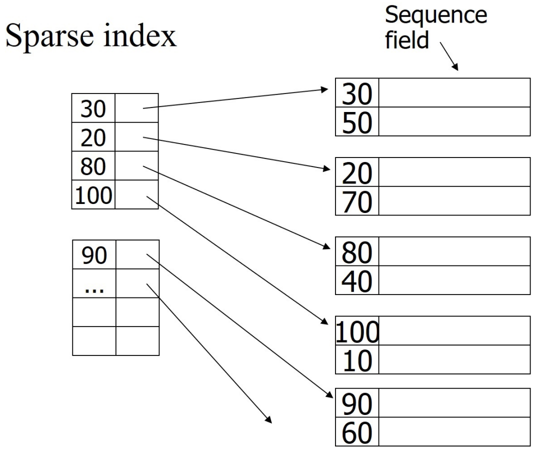
- Secondary index: an ordered index whose search key is `not` the sort key used for the sequential file



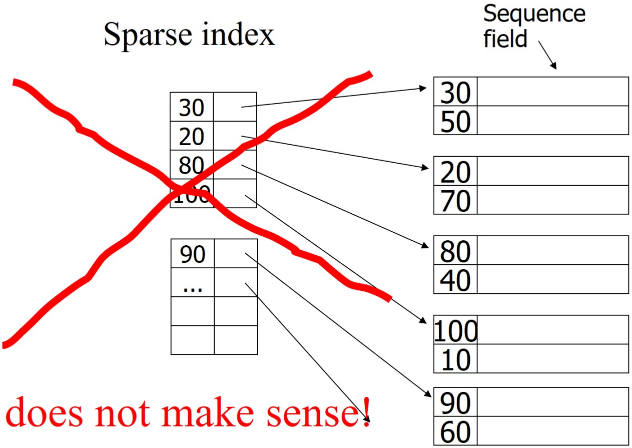
Sparse Secondary Index?



Sparse Secondary Index?



Sparse Secondary Index?



does not make sense!

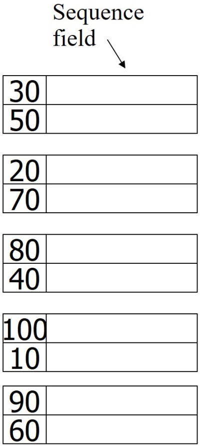
Sparse Secondary Index?

- No!
- Since records are not sorted on that key, cannot predict the location of a record from the location of any other record.
  - Can't use it to predict the location of any record not mentioned in the index
  - Can't find the record without scanning the whole file
- Thus secondary indexes are always **dense**.

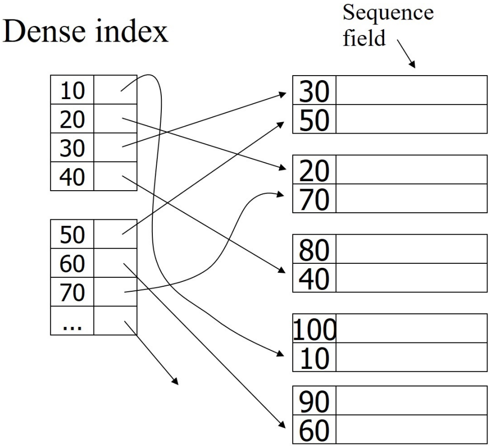
Design of Secondary Indexes

- Always **dense**, usually with duplicates
- Consists of `key-pointer` pairs ("key" means `search key`, not relation key)
- Entries in index file are sorted by `key` value
- Therefore second-level index is **sparse** (if we wish to place a second level of index)

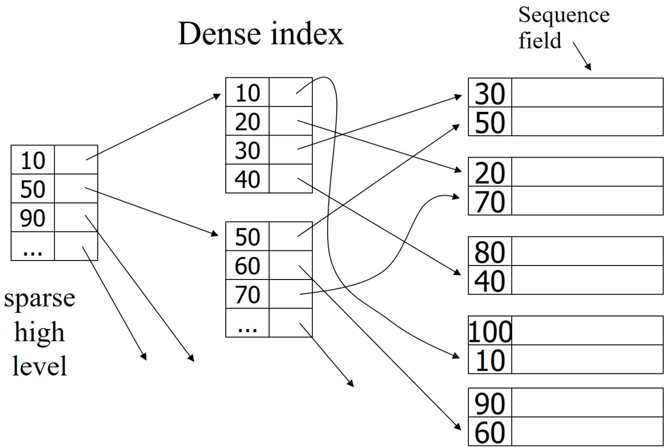
Secondary indexes



Secondary indexes



Secondary indexes

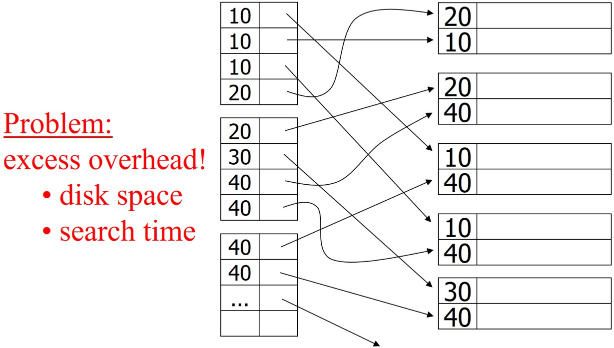


Secondary indexes

- Lowest level is **dense**
- Other levels are **sparse**
- Also: Pointers are record pointers (not block pointers)

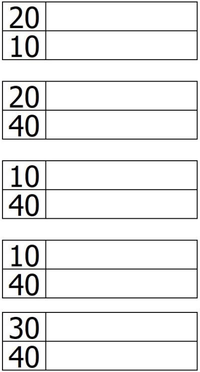
Secondary Index and Duplicate Keys

- Duplicate search keys may introduce overhead, both space and search time
- One option:



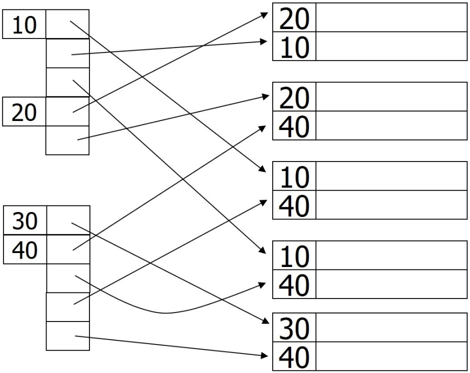
  - Waste space in the present of duplicate keys
  - If a `search key` value appears n times in the data file, then there are n entries for it in the index.

- Another option: Variable sizes index fields



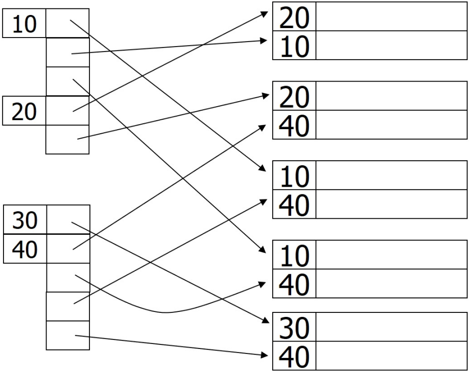
Secondary Index and Duplicate Keys

- Another option: Variable sizes index fields



Secondary Index and Duplicate Keys

- Another option: Variable sizes index fields



Problem:
variable size
records in
index!

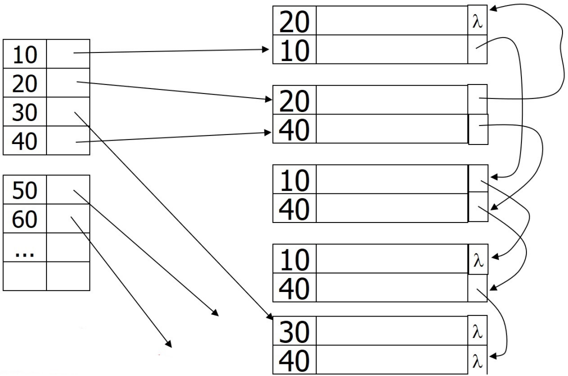
☺ saves space in index

☹ complex design and search

- *Implementation Complexity: Managing variable-sized fields requires more complex data structures to efficiently locate and manipulate index entries.*

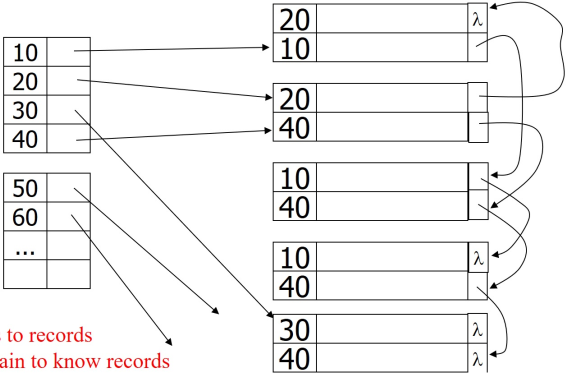
Secondary Index and Duplicate Keys

- another idea: Chain records with same key option
  - *Instead of creating separate index entries for each record with a duplicate key, the database system can chain these records together.*
  - In the index entry, a pointer or reference can point to the first record with the matching key.
  - Records with the same key are linked together in a linked list structure, where each record points to the next record with the same key.



Secondary Index and Duplicate Keys

- another idea: Chain records with same key option



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

☺ simple index, easy search

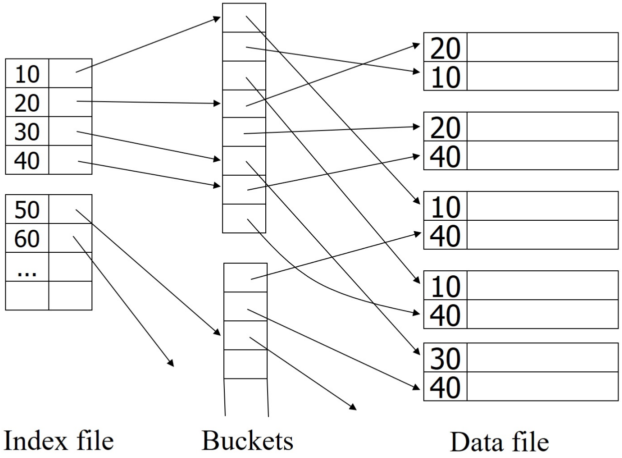
☹ add fields to records header

☹ follow chain to successive records

Buckets

- To avoid repeating values, use a level of indirection
- Instead of directly pointing to records with the same search-key value, the index entry points to a bucket.
- Buckets are intermediate data structures placed between the *secondary index file* and the *data file*.
- Each index record now points to a specific bucket that contains pointers to all actual records with the same search-key value.
- Buckets serve as a convenient way to avoid storing duplicate values directly in the index.
  - The system designates a specific file to store these buckets.
  - Each index entry for a search key `K` points to the first element in the corresponding bucket for search key `K`.
  - The bucket, in turn, holds pointers or references to all records with search key `K`.
  - The designated bucket file is managed similarly to other sorted files in the database.

Secondary Index and Duplicate Keys



Why "bucket" idea is useful

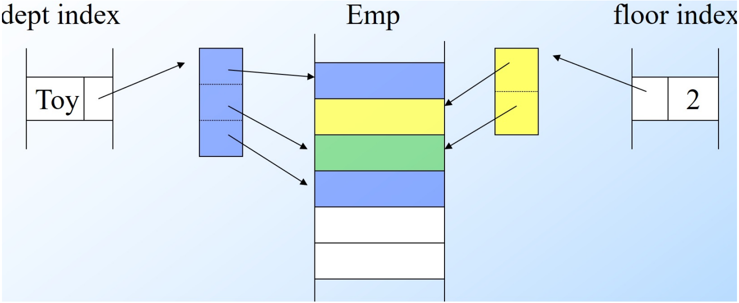
- We can use the pointers in the buckets to help answer queries without ever looking at most of the records in the data file.
  - When there are several conditions to a query, and each condition has a secondary index to help it,
  - find the bucket pointers that satisfy all the conditions by intersecting sets of pointers (in memory), and
  - retrieving only the records pointed to by the surviving pointers.
- Save the I/O cost of retrieving records that satisfy some, but not all, of the conditions

Why “bucket” idea is useful

- Consider the relation **Emp**`(name, dept, floor)`
- Suppose we have a **primary index** on `name`, **secondary indexes** with **indirect buckets** on both `dept` and `floor`.
- Query:

```sql
SELECT name
FROM   Emp
WHERE  dept = 'Toy' AND floor = 2;
```

Query: Get employees in (Toy Dept) & (2nd floor)



- Intersect `Toy dept` bucket and `floor 2` bucket to get set of matching **Emp**'s
- Retrieving the minimum possible number of data blocks. Saves disk I/O's

Summary so far

- Conventional index
  - Basic Ideas: sparse, dense, multi-level
  - Duplicate Keys
  - Deletion/Insertion
  - Secondary indexes

Conventional indexes

- Conventional indexes are simple to implement and good for scans, but they have several disadvantages, including:
  - Unbalanced performance, with the number of operations to find a record varying depending on the data
  - Degraded performance as the file grows, due to the creation of overflow blocks
  - The need for periodic reorganization of the entire file
  - Not being widely used in commercial systems
- `B+-Tree` indexes are a more efficient alternative, automatically reorganizing themselves with small, local changes in response to insertions and deletions.
- This eliminates the need for periodic reorganization and maintains performance as the file grows.
- However, `B+-Tree` indexes do have some disadvantages, including:
  - Increased overhead for insertions and deletions
  - Increased space overhead (store additional information to support their efficient operation)
- In general, `B+-Tree` indexes are a better choice for most applications than conventional indexes.

Outline

- Conventional indexes
  - Basic Ideas: sparse, dense, multi-level …
  - Duplicate Keys
  - Deletion/Insertion
  - Secondary indexes
- `B+-Trees`
- Hash Tables