Indexing

Part I: Conventional indexes

Slides: adapted mainly from CS525 by Yousef M. Elmehdwi, Illinois Institute of Technology

This Chapter: File Organization

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

How to Represent a Relation

- 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

Indexes

- Use indexes
  - special data structures
  - help to retrieve data quicker for certain queries
  - make the search for records based on certain fields, called indexing fields, more efficient.
- Possible data structures
  - simple indexes on sorted files
  - secondary indexes on unsorted files
  - B-trees
  - hash table

- **Search key**: single attribute type, or set of attribute types, whose values determine criteria according to which records are retrieved
  - can be primary key, alternative key, or one or more non-key attribute types
  - can be composite, e.g. (country, gender)
  - can also be used to specify range queries, e.g. YearOfBirth between 1980 and 1990

Introductory Concepts

- **Primary file organization methods**: determine physical positioning of stored records on storage medium
  - E.g., heap files, random file organization, indexed sequential file organization
    - When implementing a physical file, the records are physically organized according to the primary file organization method
  - can only be applied once
- **Linear search**: on the entire file for records that match the search key: each record in file is retrieved and assessed against search key
- **Hashing and indexing**: primary techniques that specify relationship between record's search key and physical location

- **Secondary file organization methods**: provide constructs to efficiently retrieve records according to search key that was not used for primary file organization
  - based on secondary index

Heap File Organization

- Basic primary file organization method
- New records inserted at end of file
- No relationship between record's attributes and physical location
- Only option for record retrieval is linear search
- For a file with $\mathbb{B}$ blocks, it takes on average $\frac{B}{2}$ sequential block accesses to find record according to unique search key
- Searching records according to non-unique search key requires scanning entire file

Sequential File Organization

- Records stored in ascending/descending order of search key
- Efficient to retrieve records in order determined by search key
- Records can still be retrieved by means of linear search, but now a more effective stopping criterion can be used, i.e. once first higher/lower key value than required one is found
- Expected number of block accesses to retrieve record according to primary key by means of
  - linear search: $B/2$ sequential block accesses
  - binary search: $\log_2 B$ random block accesses

Sequential File Organization

Number of records (NR)	30000
Block size (BS)	2048 bytes
Records size (RS)	100 bytes

- BF (Blocking Factor/No of records per block) = ⌊BS/RS⌋ =⌊2048/100⌋=20
- NBLK (No of Blocks) =30000/20=1500
- If single record is retrieved according to primary key using linear search, expected number of required block accesses is
- 1500/2 = 750 sba
- If binary search is used, expected number of block accesses is log2(1500) ≈ 11 rba

Random File Organization (Hashing)

- Random file organization (a.k.a. direct file organization, hash file organization) assumes direct relationship between value of search key and physical location
- Hashing algorithm defines key-to-address transformation
  - generated addresses pertain to bucket (contiguous area of record addresses)
- Most effective when using primary key or other candidate key as search key

Random File Organization (Hashing)

- Hashing cannot guarantee that all keys are mapped to different hash values, hence bucket addresses
- Collision occurs when several records are assigned to same bucket (also called synonyms) If more synonyms than slots for a bucket, bucket is in overflow
  - additional block accesses needed to retrieve overflow records
- Hashing algorithm should distribute keys as evenly as possible over the respective bucket addresses

Random File Organization (Hashing)

- Popular hashing technique is division:

$$\texttt{address}(\texttt{key}_i) = \texttt{key}_i \bmod \texttt{M}$$

  - M is often a prime number (close to, but a bit larger than, the number of available addresses)

Indexed Sequential File Organization

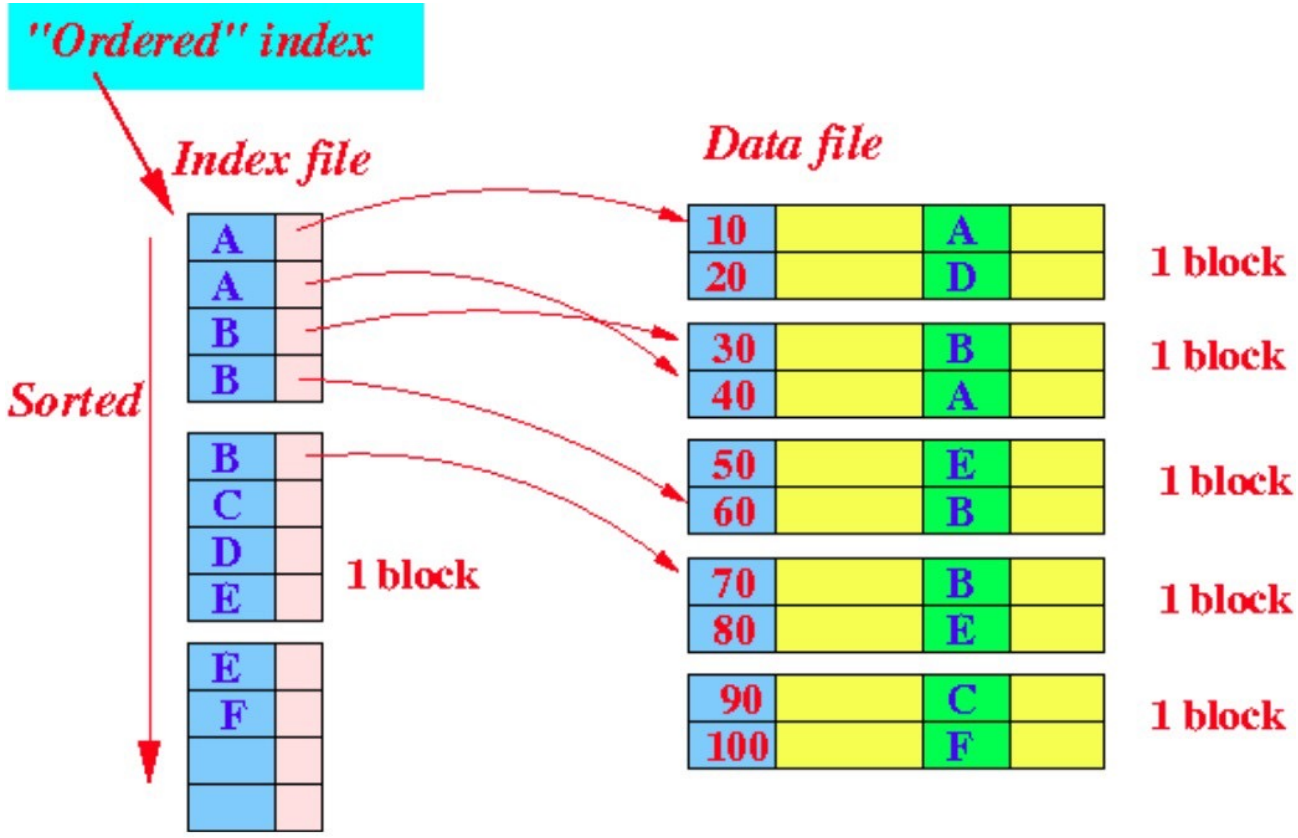
- Random file organization is efficient to retrieve individual records by search key value
- Sequential File Organization is efficient if many records are to be retrieved in certain order
- Indexed Sequential File organization method reconciles both concerns
- Indexed Sequential File organization combines sequential file organization with one or more indexes

Index

- `Index`: a data structure that enable the user to find (locate) data items efficiently (quickly) using search keys
- An index file is a file containing **key-pointer** pairs of the form `(K,a)`
- `K` is a **search key**, `a` is an **address/pointer to a block/record**
  - The record at **address** `a` has **search key** `K`
- Particularly useful when the **search key** is the **primary key** of the relation
- The size of an index file is usually much smaller than the size of a data file
- The pointer is usually a block pointer
  - So the index allow you to locate the block that contain the record quickly
- The record is found by a search operation inside the block (after the block is read into main memory)

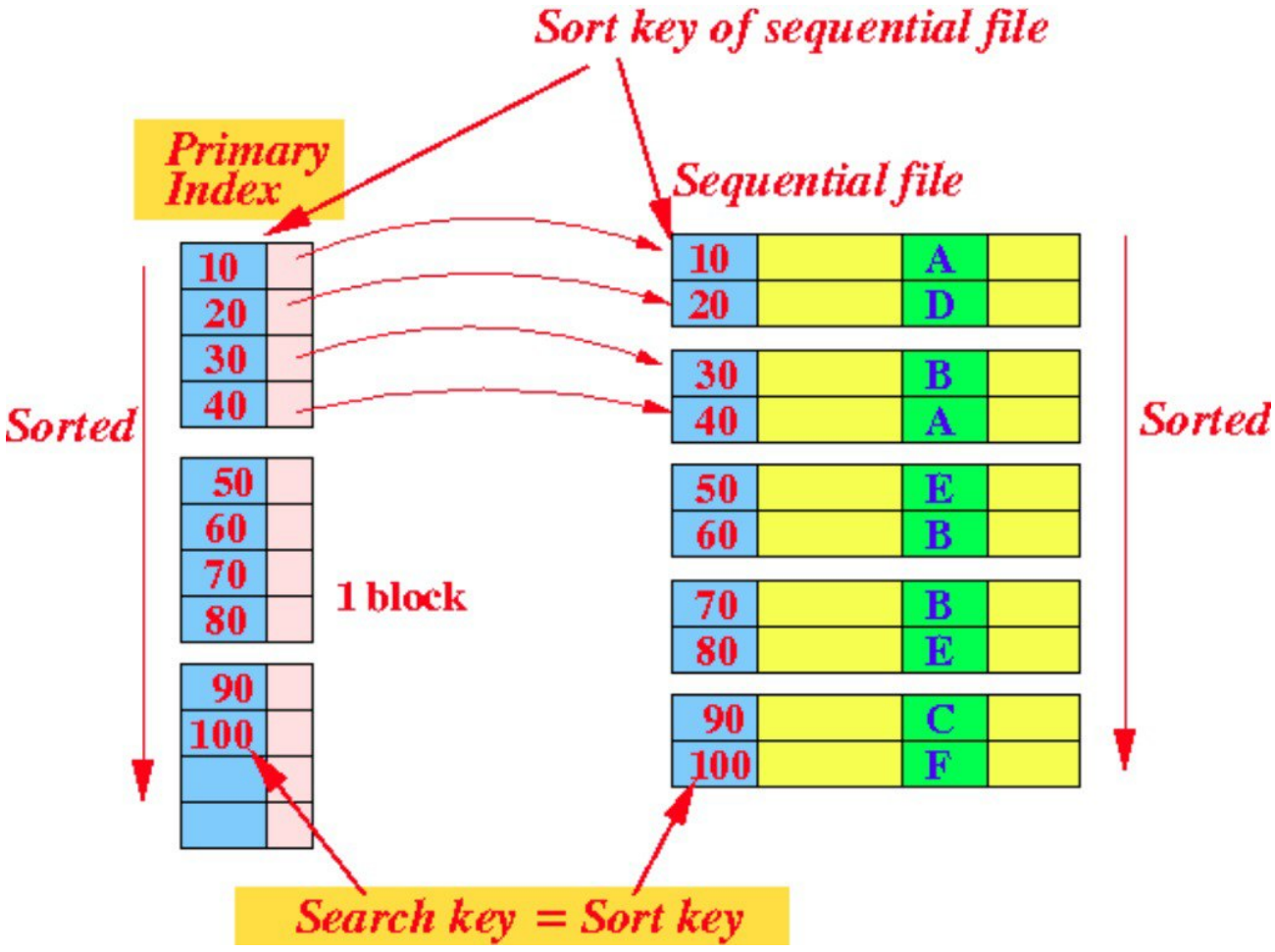
Index: Ordered Index

- `Ordered index`: an index file where the index entries are sorted (in the order of the search key)



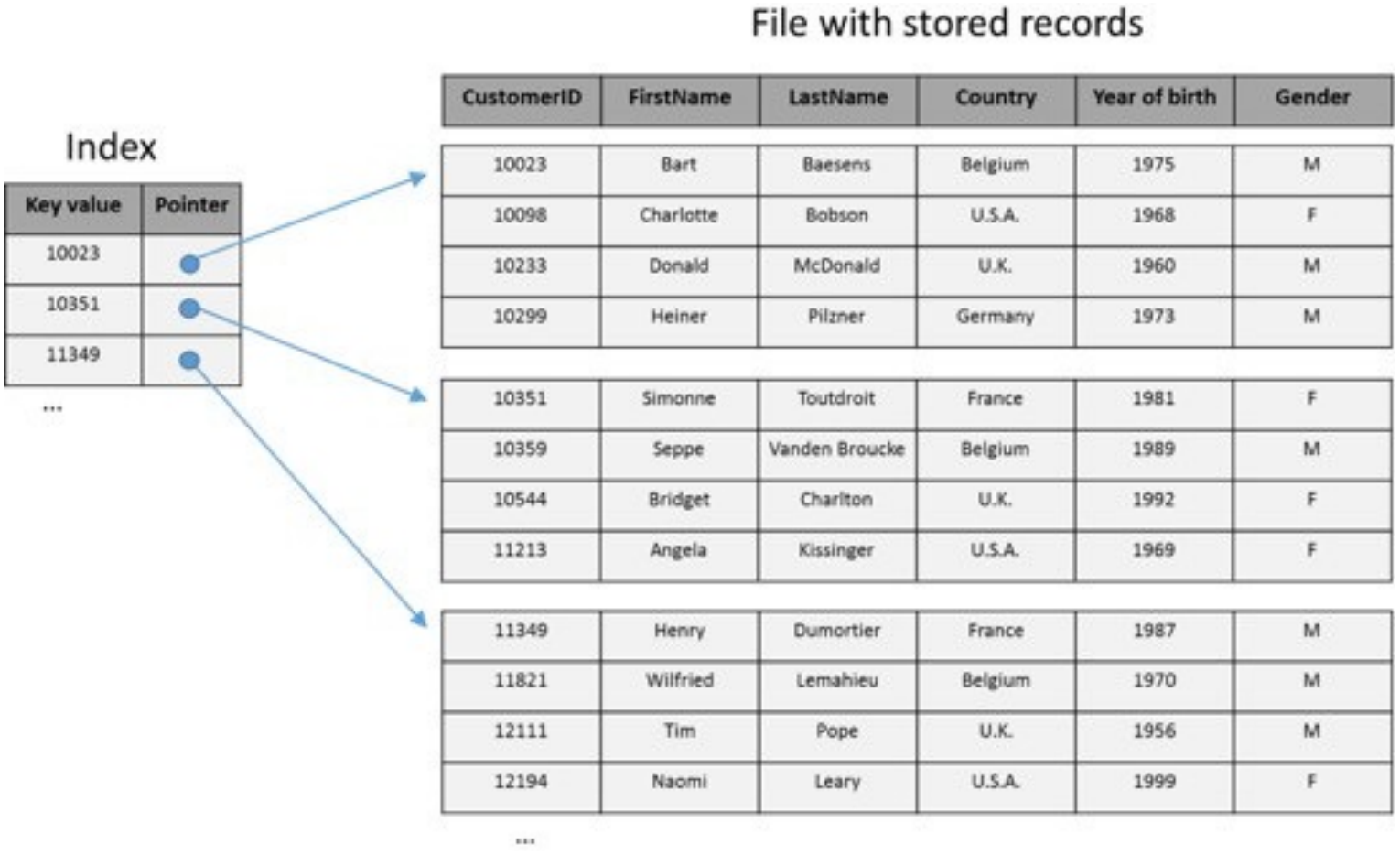
Index: Primary index

- `Primary index`: an ordered index whose search key is also the sort key used for the sequential file
  - `Sort key`: field(s) whose values are used to sort/order the records in a sequential file



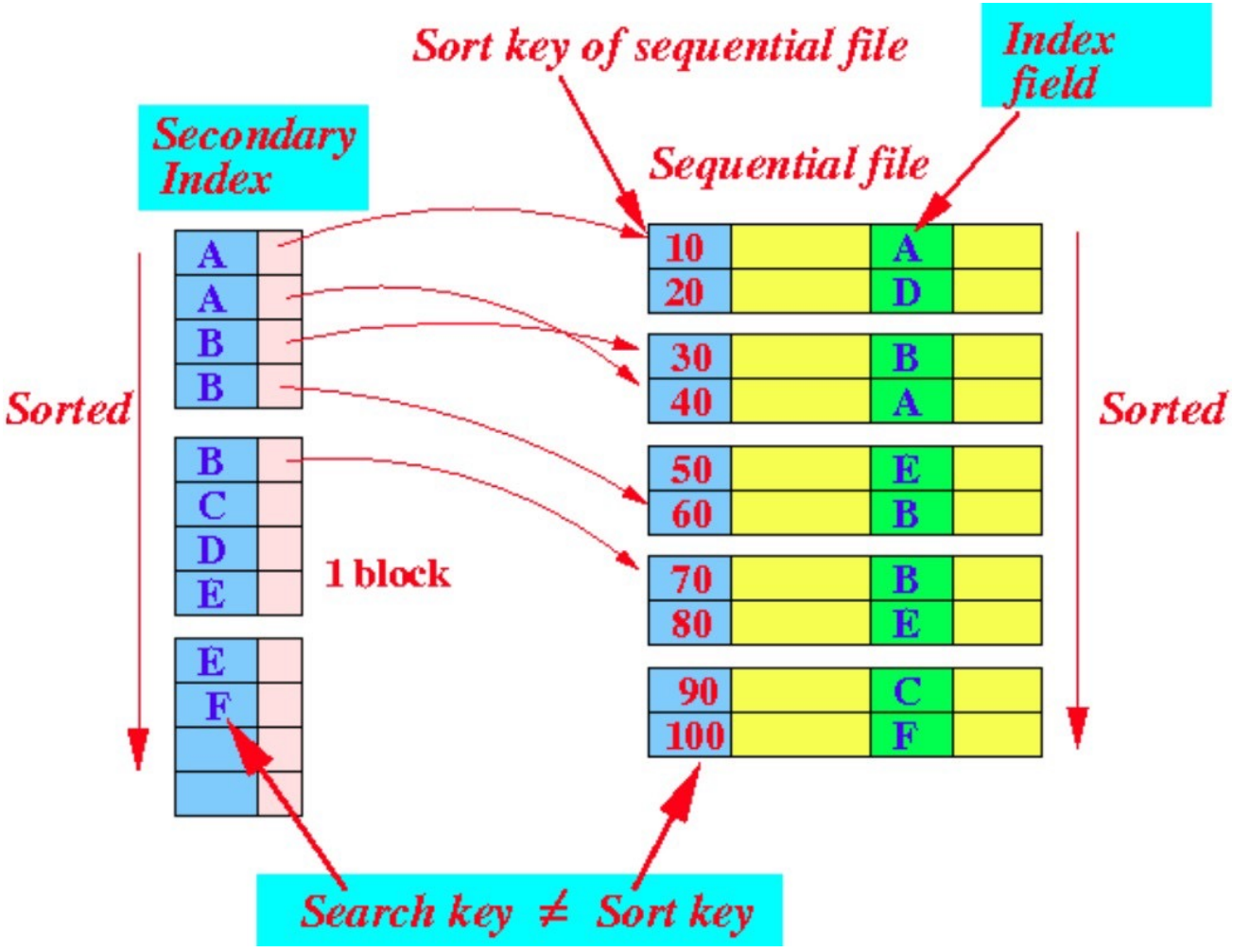
Index: Primary index: Example

- With primary index file organization, data file is ordered on unique key and index is defined over this unique search key



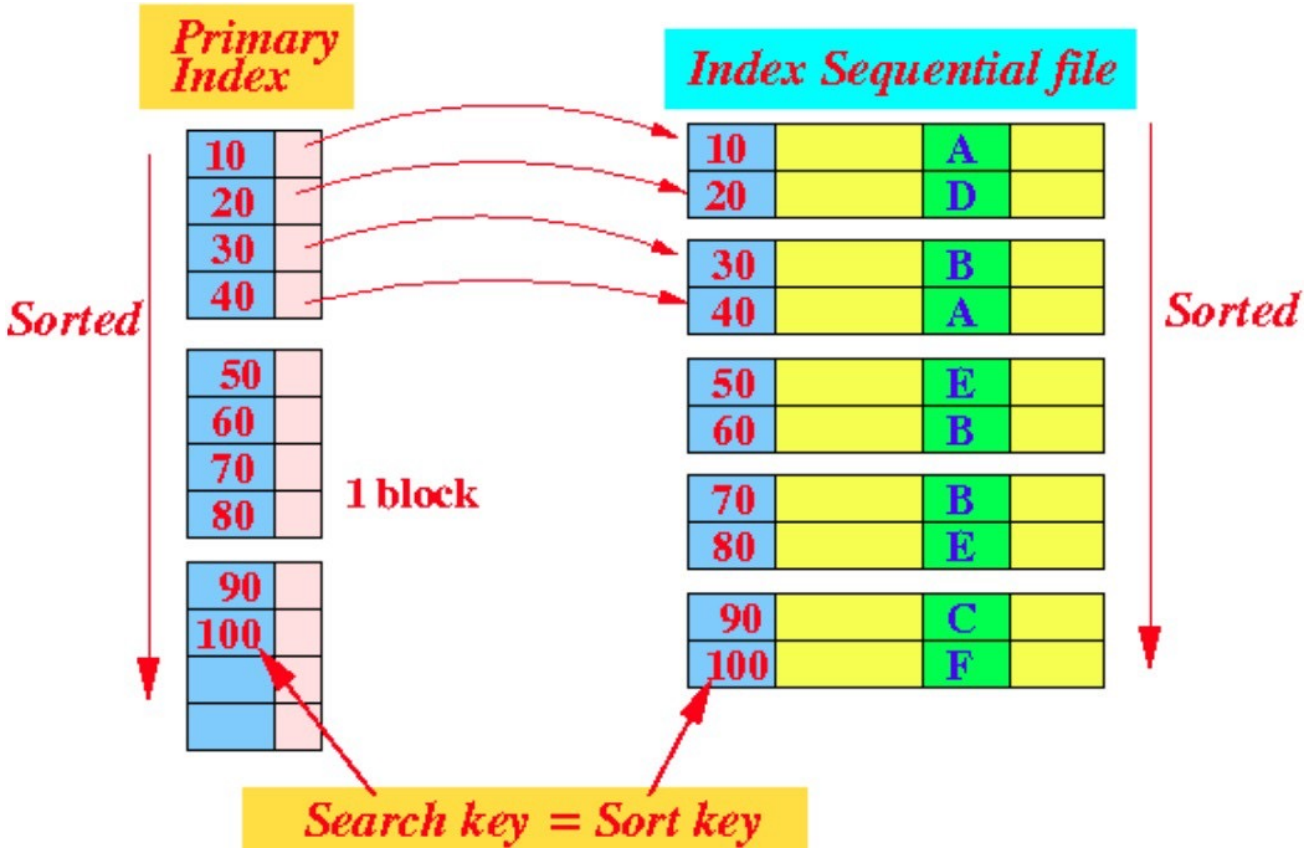
Index: Secondary index

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



Index sequential file

- `Index sequential file`: a sequential file that has a primary index



Indexed Sequential File Organization

- **Dense index** has index entry for every possible value of search key
- **Sparse index** has index entry for only some of search key values
- 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

Indexed Sequential File Organization

Linear search	NBLK sba
Binary search	$\log_2$(NBLK) rba
Index based search	$\log_2$(NBLKI) + 1 rba, with NBLKI << NBLK

Note: NBLKI represents the number of blocks in index

Indexed Sequential File Organization

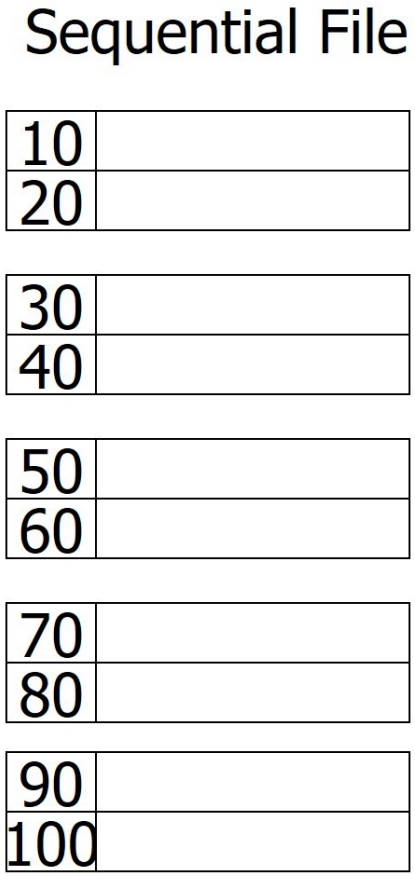
Number of records (NR)	30000
Block size (BS)	2048 bytes
Records size (RS)	100 bytes
Index entry	15 bytes

- Blocking factor of index = ⌊2048/15⌋=136
- NBLKI = ⌈1500/136⌉ = 12 blocks
- Binary search on index requires $\log_2(12) + 1 \approx 5$ rba
- (compare to 750 sba and 11 rba!)

Indexes on Sequential Files: Example of a 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

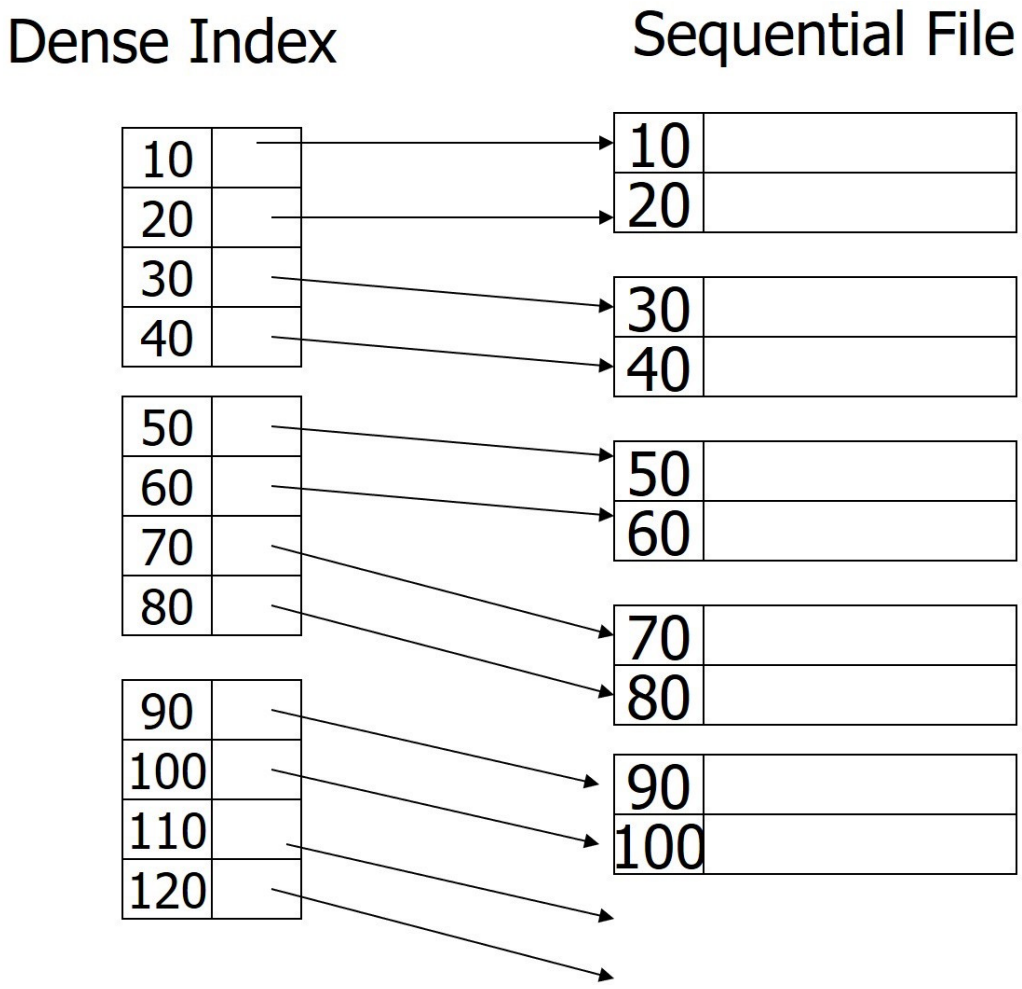


Indexes on Sequential Files: Example of a Dense Index

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

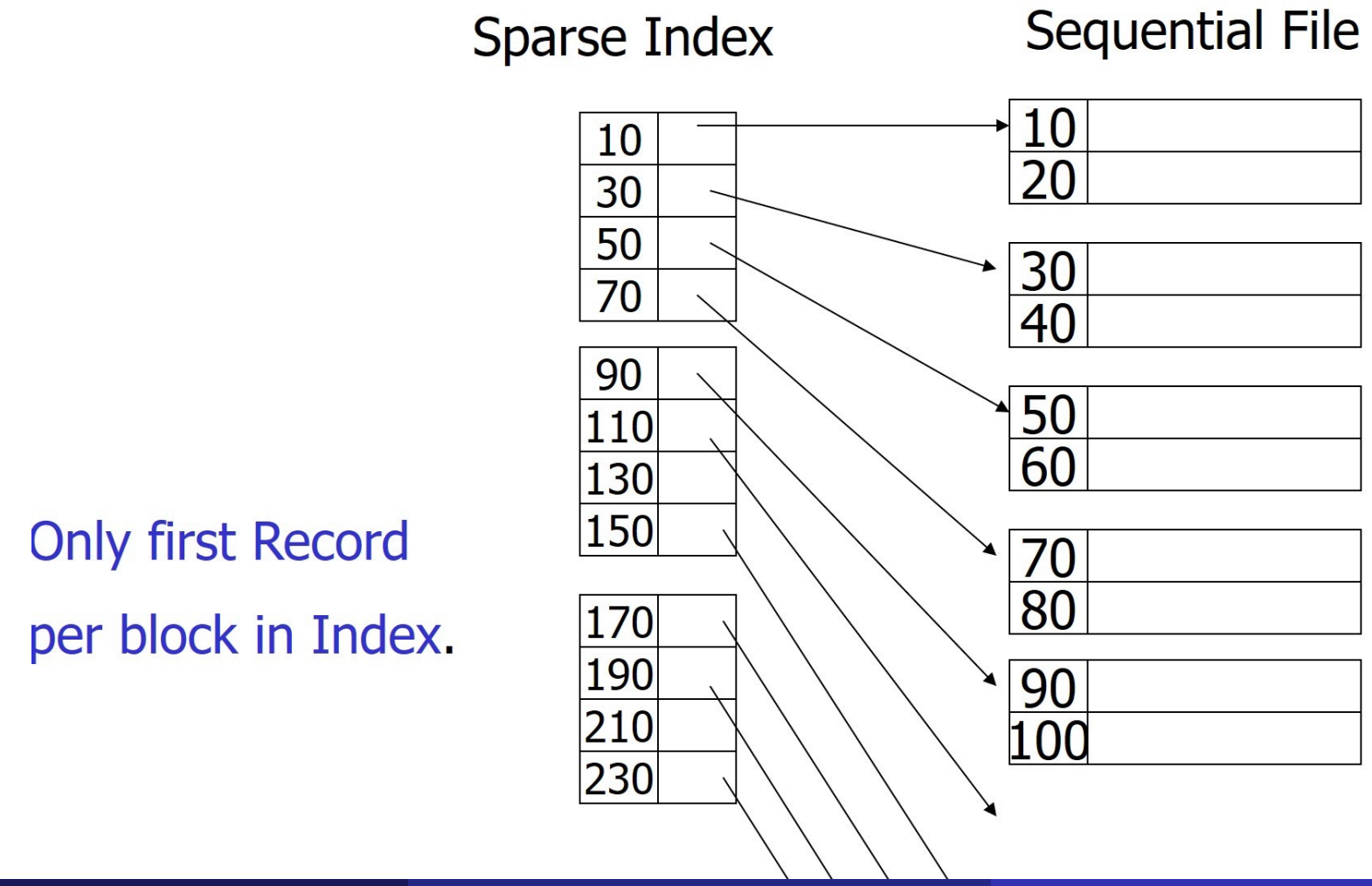
Every Record is in Index.



Indexes on Sequential Files: Sparse Index Example

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.

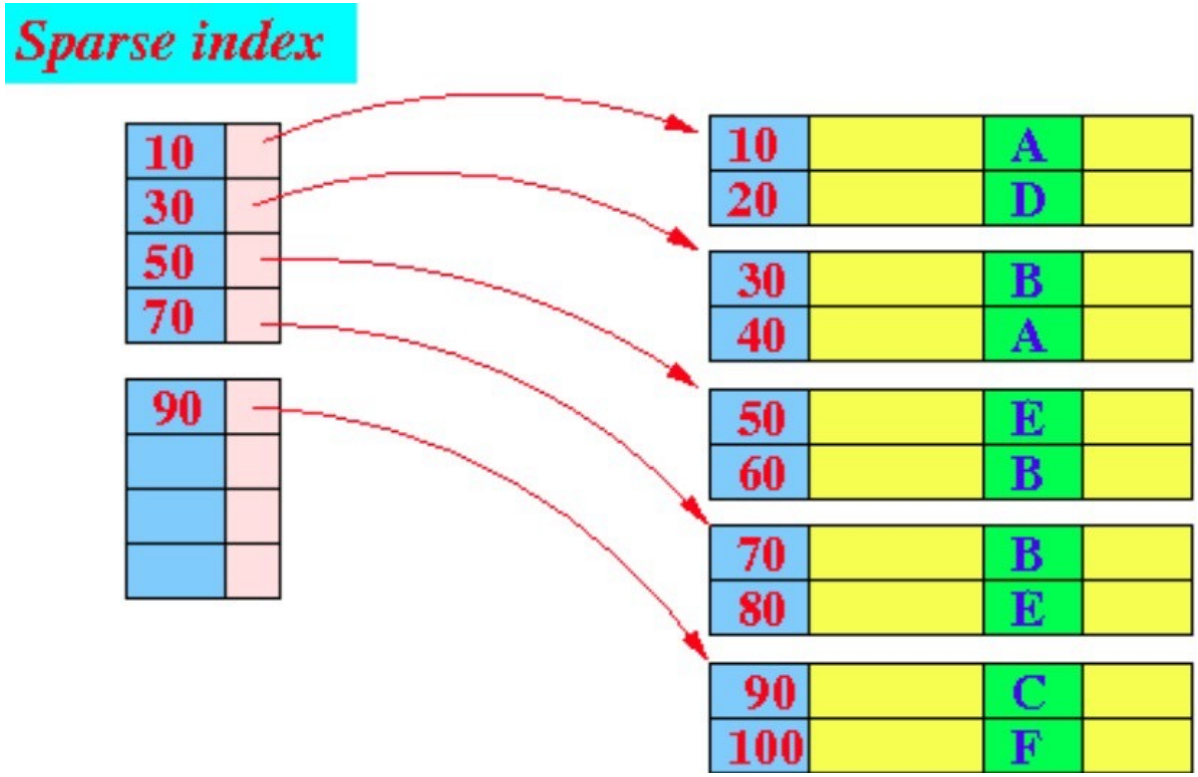


Indexes on Sequential Files: Using a Sparse Index

- To locate a record with `key` K:
  1. Find the index record with the largest search-key $\leq K$
  2. Retrieve the indicated data block
  3. Search the block for the record with `key` K

Indexes on Sequential Files: 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

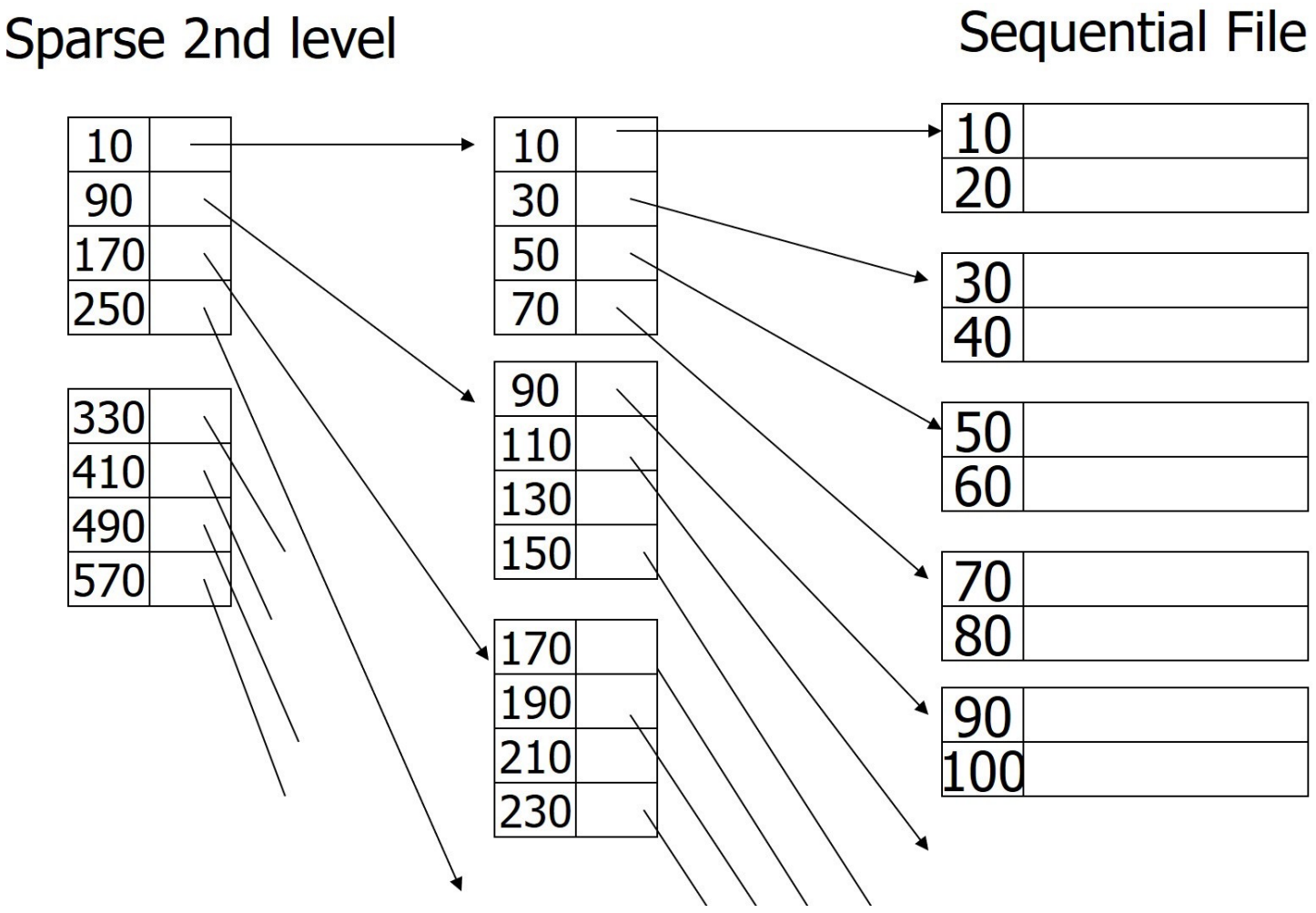
Comment

- Sparse indices are uncommon
- Because
  - Only a primary index can be a sparse index
  - This requires that the file is sorted on the search key
- Most commonly used indexes is: secondary index
- Very flexible: The file does not need to be sorted

Multi-level Index

- If primary index does not fit in memory, access becomes expensive.
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - **outer index** - a sparse index of primary index
  - **inner index** - the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.

Two-Level Index Example



Comment

- *{*`FILE,INDEX`*}* may be layed out on disk as either contiguous or blocks chained strategy

Sparse vs. Dense Tradeoff

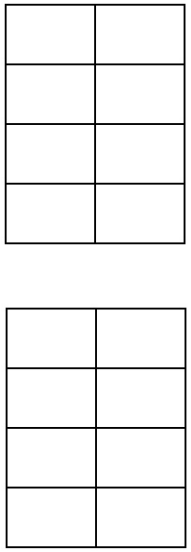
- Sparse
  - uses much less space
  - (Later: sparse better for insertions)
- Dense
  - unlike sparse, can tell if any record exists without accessing file
  - (Later: dense needed for secondary indexes)

Next

- Duplicate keys
- Deletion/Insertion
- Secondary indexes

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



10	
10	

10	
20	

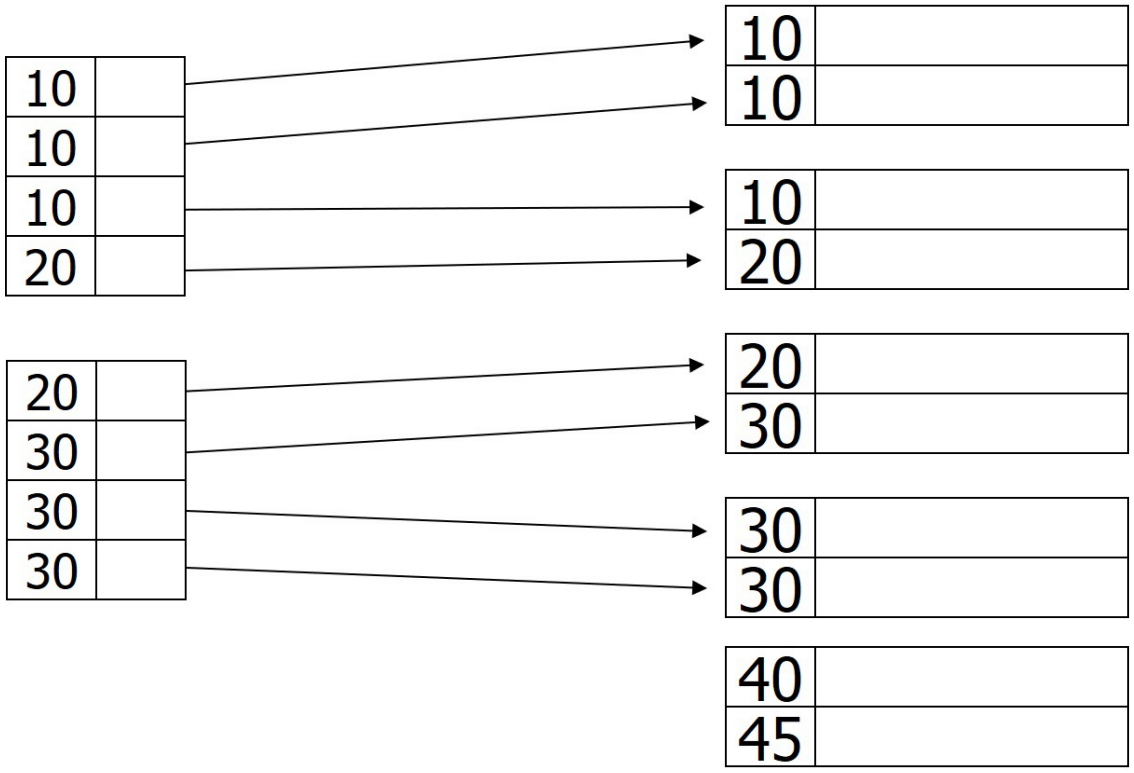
20	
30	

30	
30	

40	
45	

Duplicate Search Keys with Dense Index

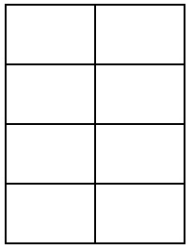
- Dense index, one way to implement? (Point to each value)
  - one entry with key *K* for each record of the data file that has search key *K*

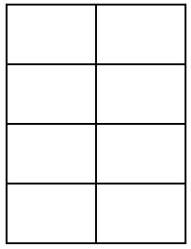


- To find all data records with search key *K*, follow all the pointers in the index with search key *K*

Duplicate Search Keys with Dense Index

- Dense index, better way?





10	
10	

10	
20	

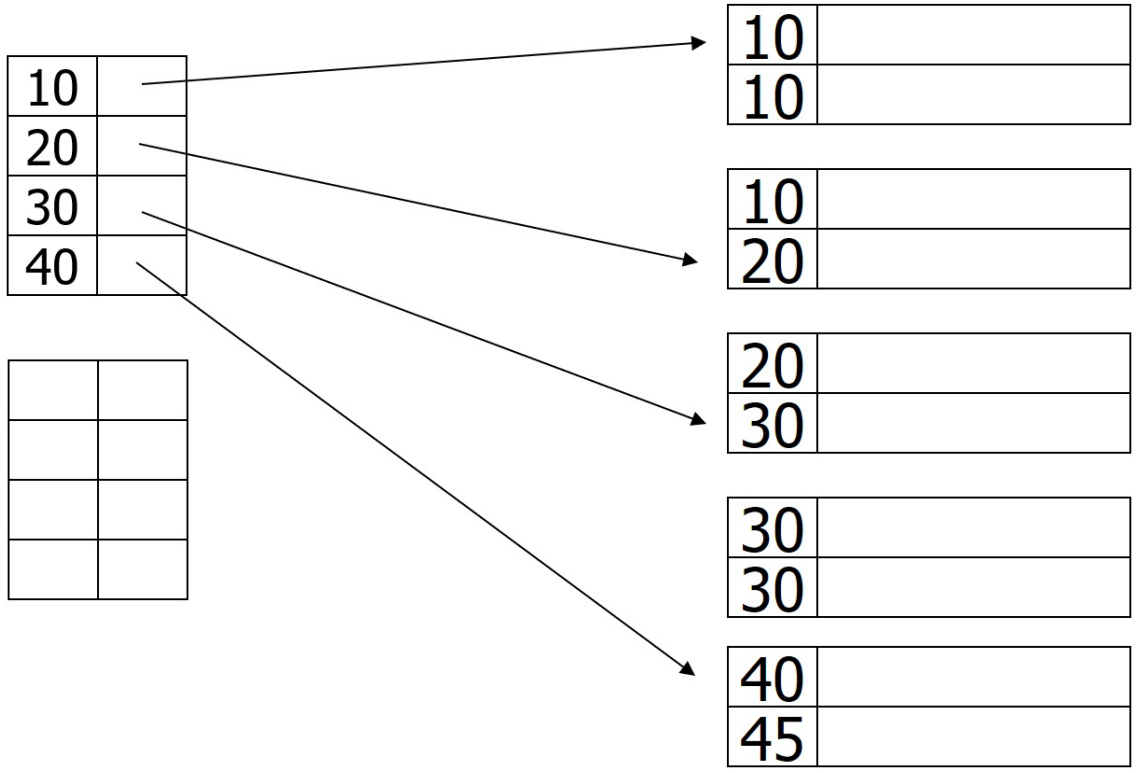
20	
30	

30	
30	

40	
45	

Duplicate Search Keys with Dense Index keys

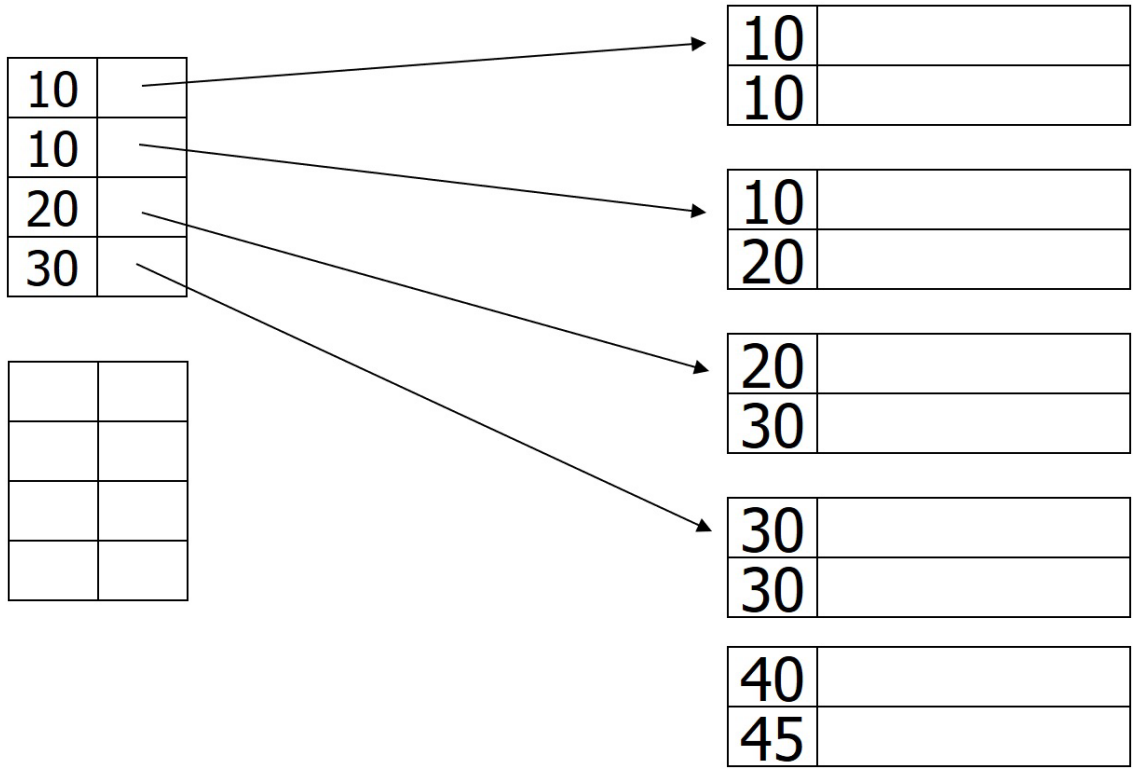
- Dense index, better way? Point to each distinct value!
  - only keep record in index for first data record with each search key value (saves some space in the index)



- 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?
  - key-pointer pairs corresponding to the first search key on each block of the data file.



- Find all records with search key 20? Search key =10?

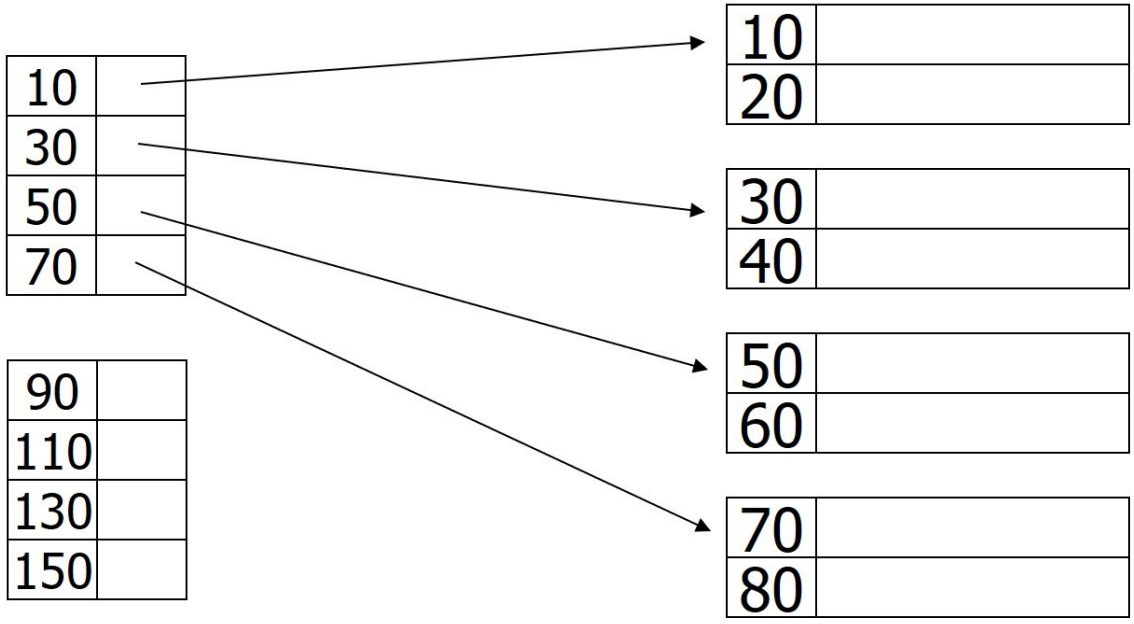
Duplicate search keys with sparse index

- To find all data records with search key K:
  - find last entry (E_1) in index with $key \leq K$
  - move towards front of index until either reaching entry (E_2) with $key < K$ or come to the 1st entry
  - check data blocks pointed to by entries from E_2 to E_1 for records with search key K

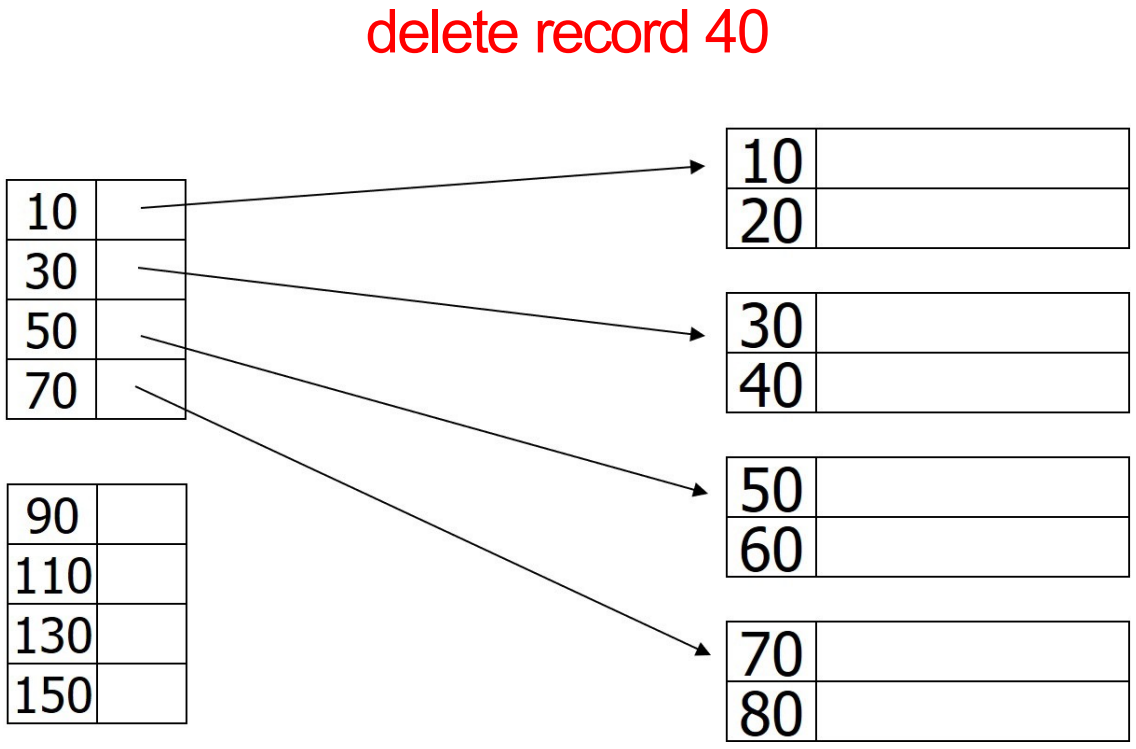
Next

- Duplicate keys
- Deletion/Insertion
- Secondary indexes

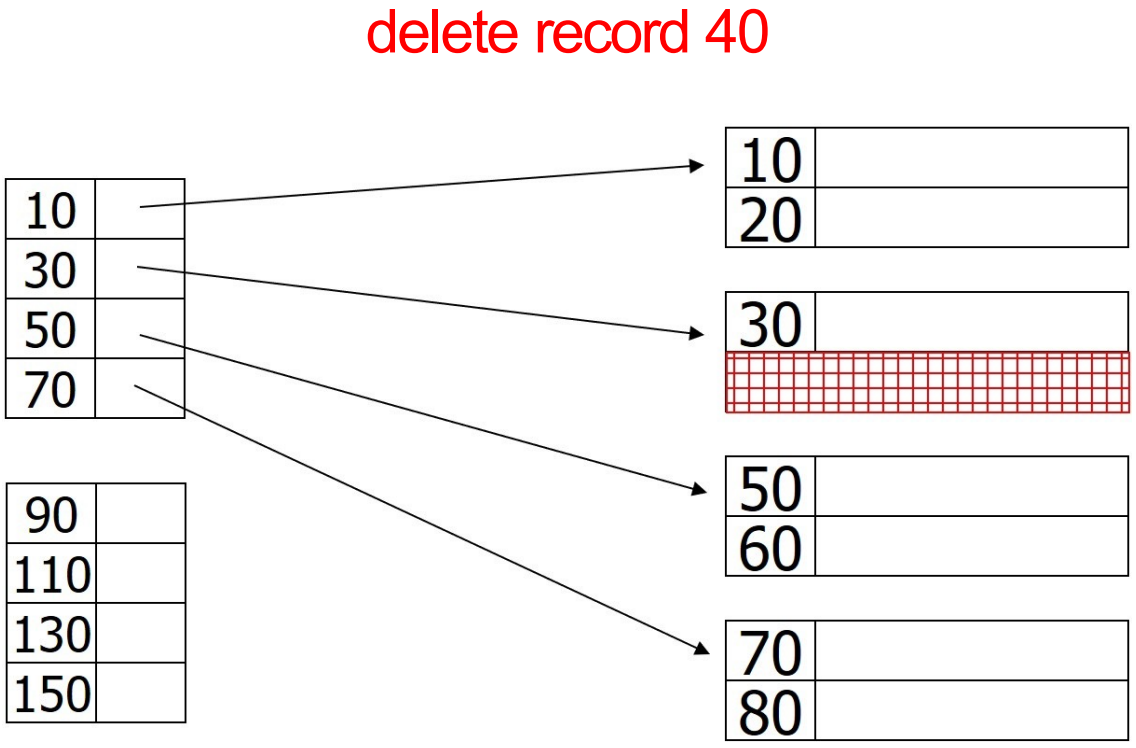
Deletion from sparse index



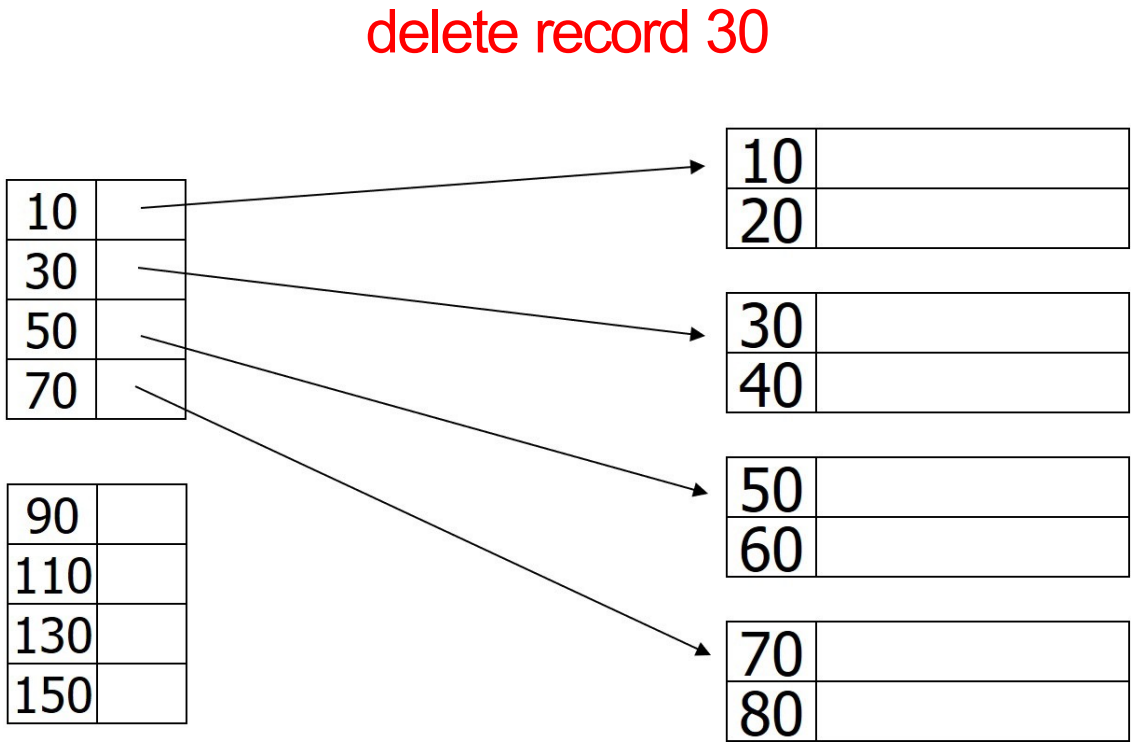
Deletion from sparse index



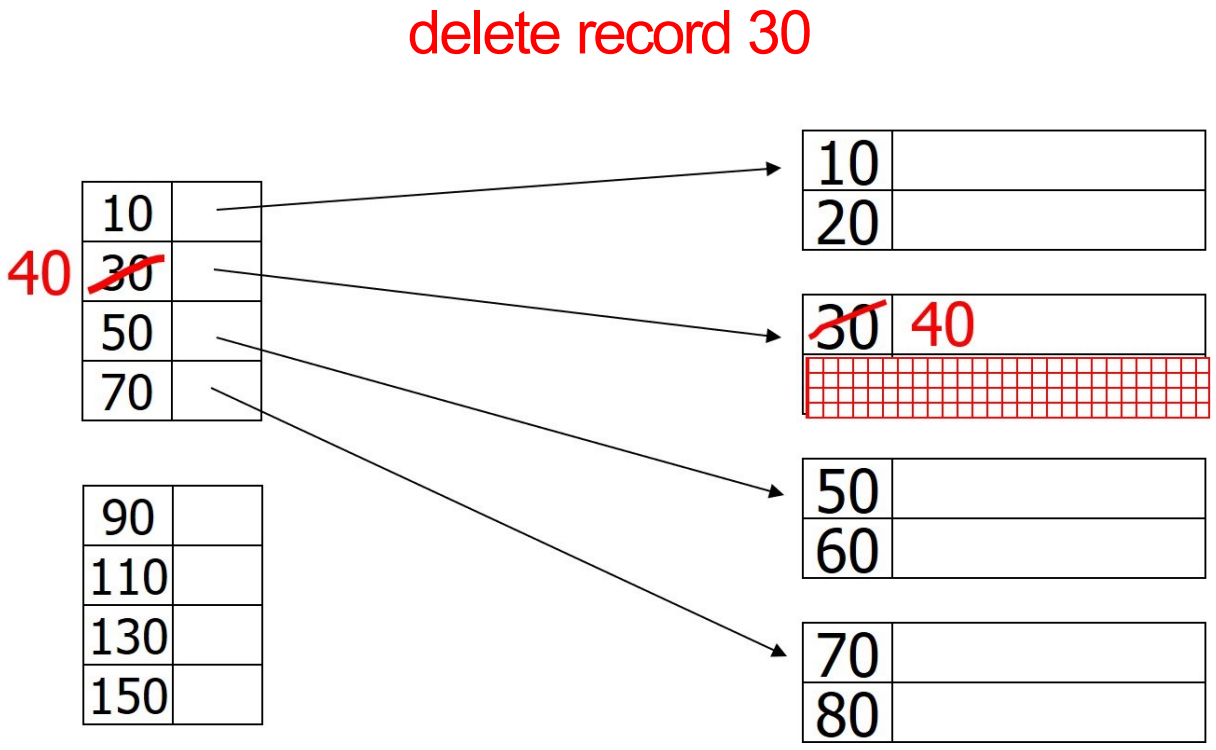
Deletion from sparse index



Deletion from sparse index

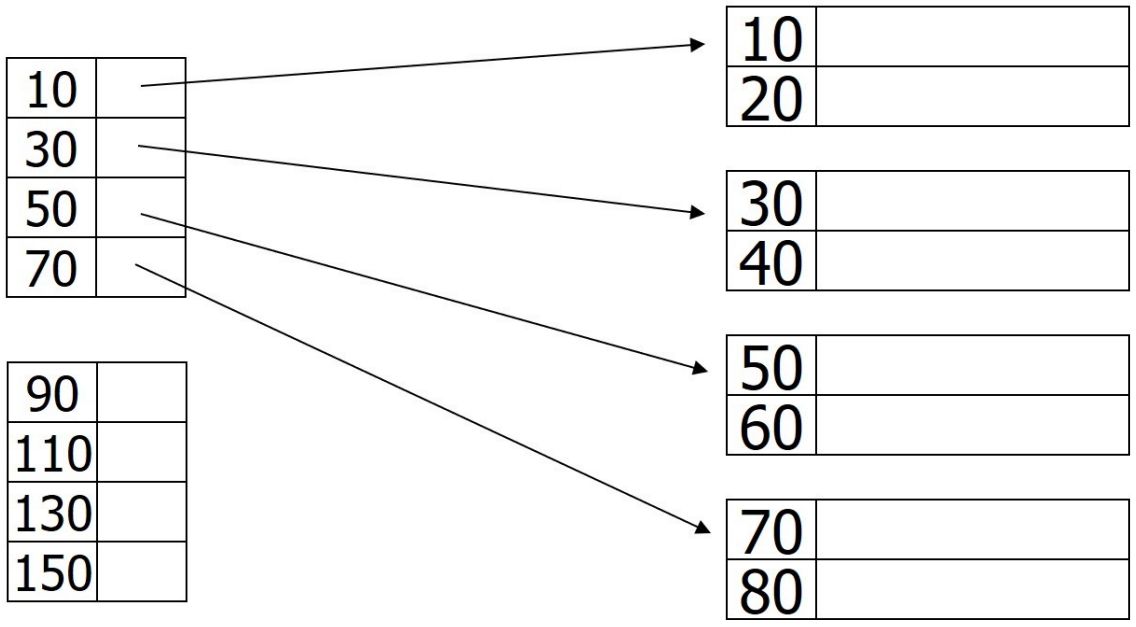


Deletion from sparse index

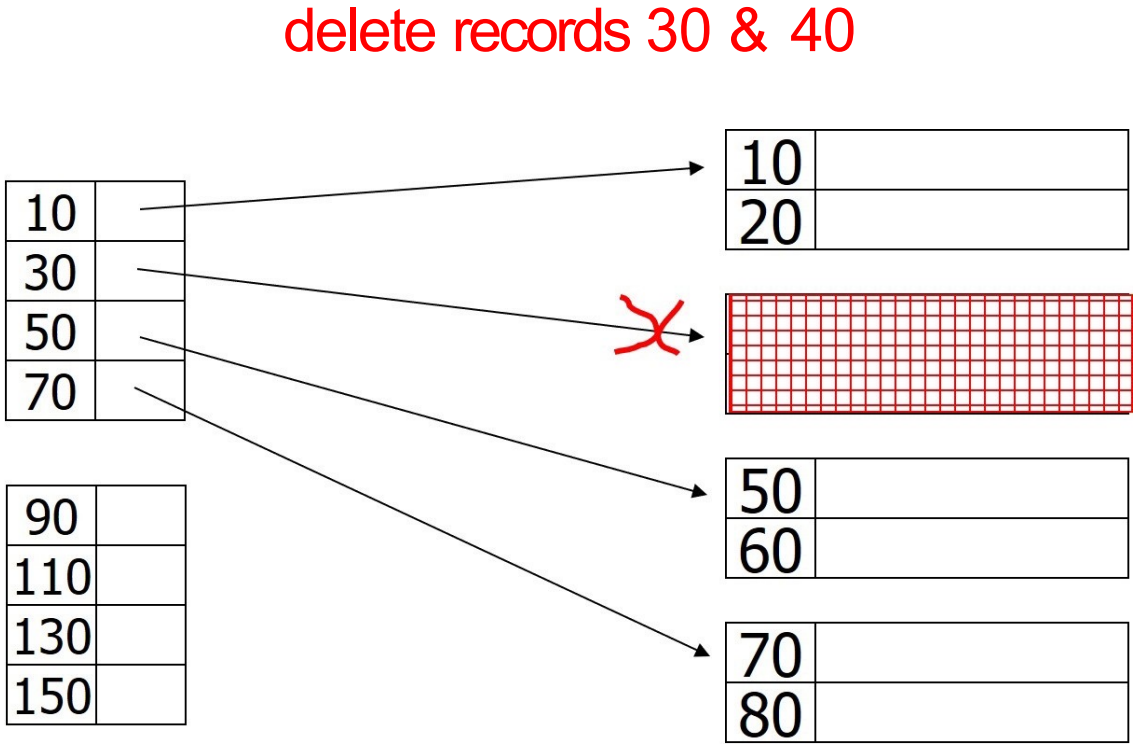


Deletion from sparse index

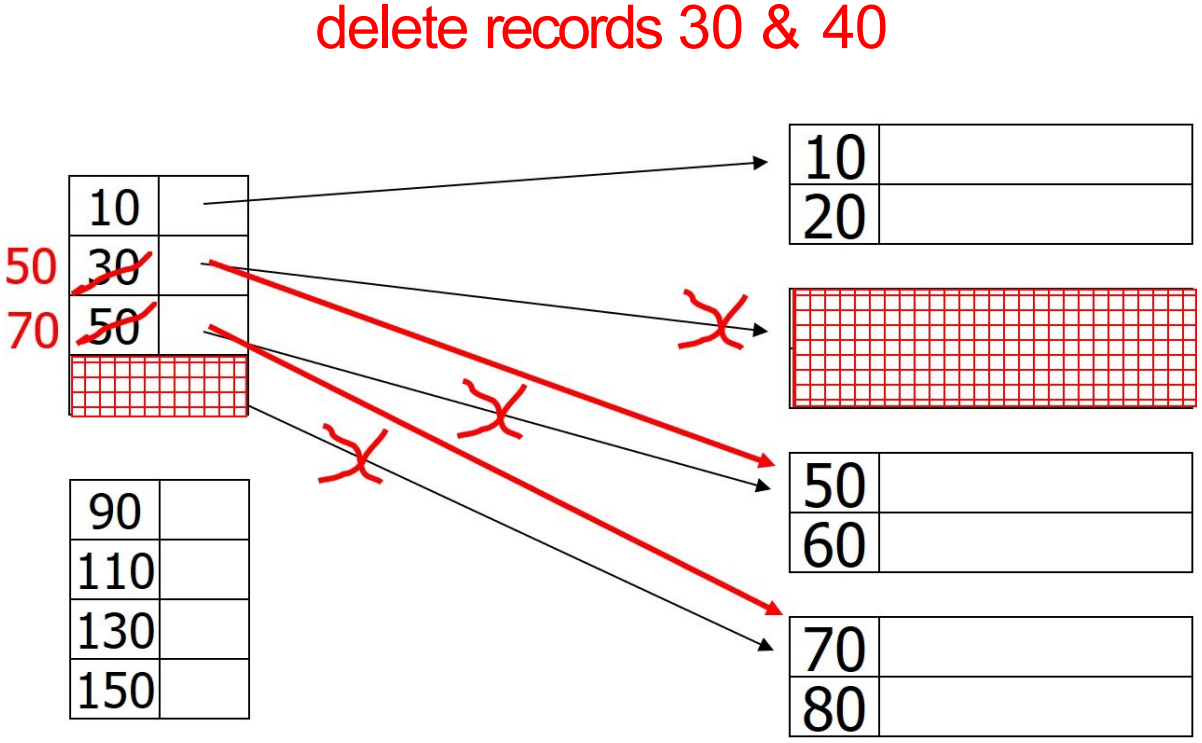
delete records 30 & 40



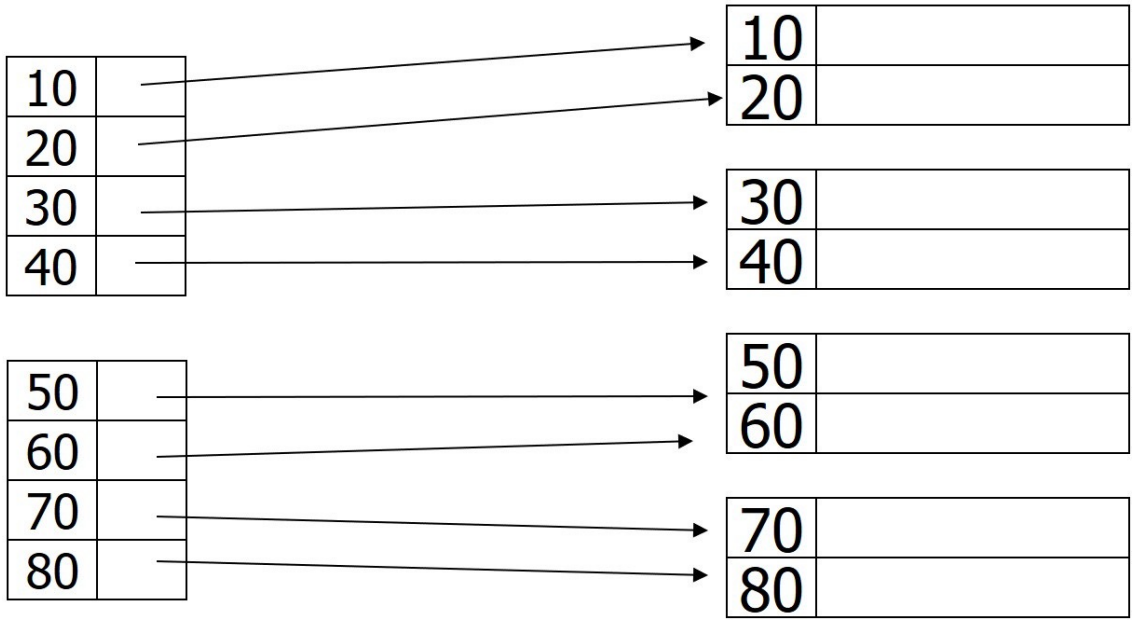
Deletion from sparse index



Deletion from sparse index

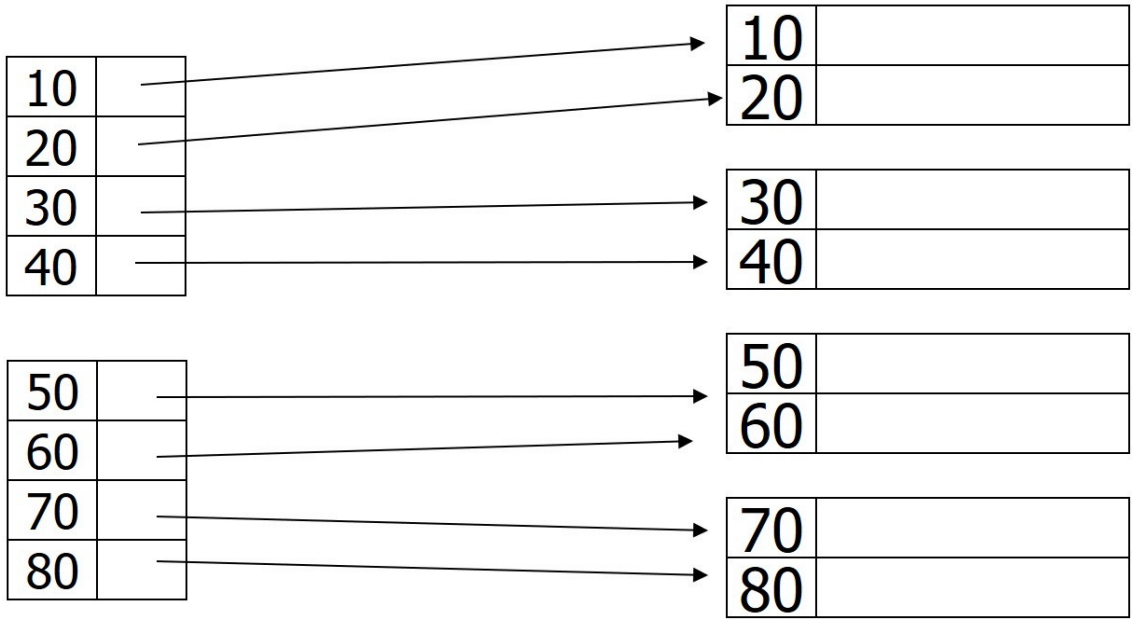


Deletion from dense index

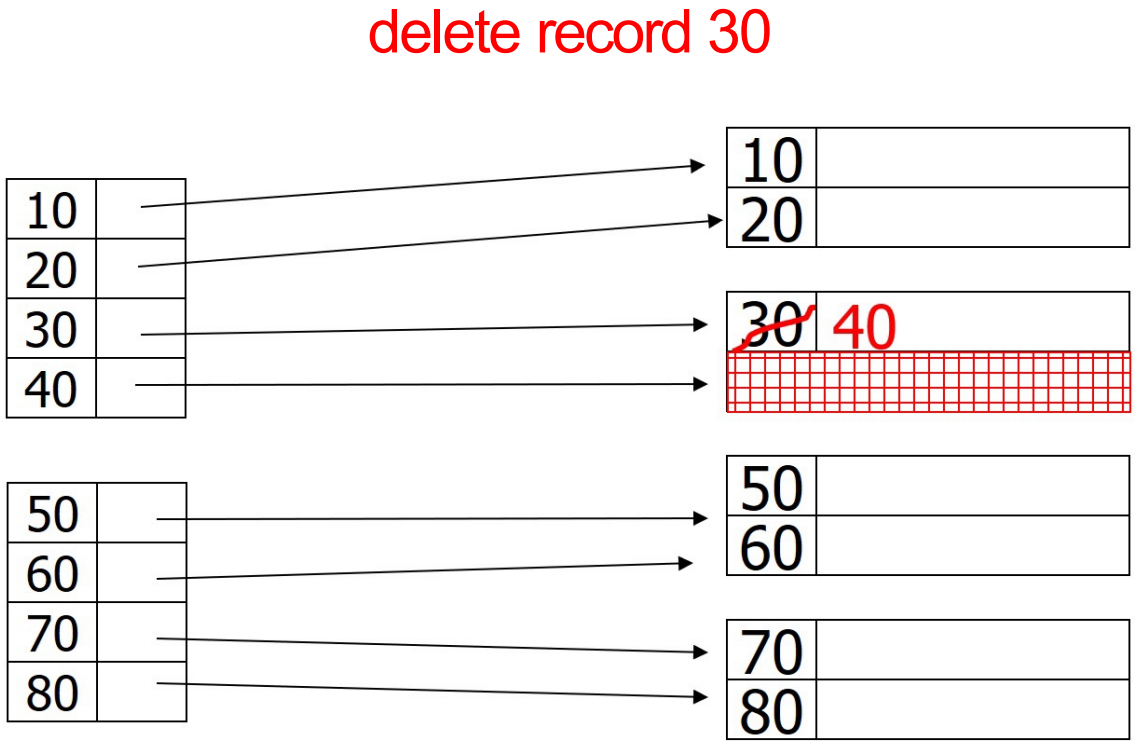


Deletion from dense index

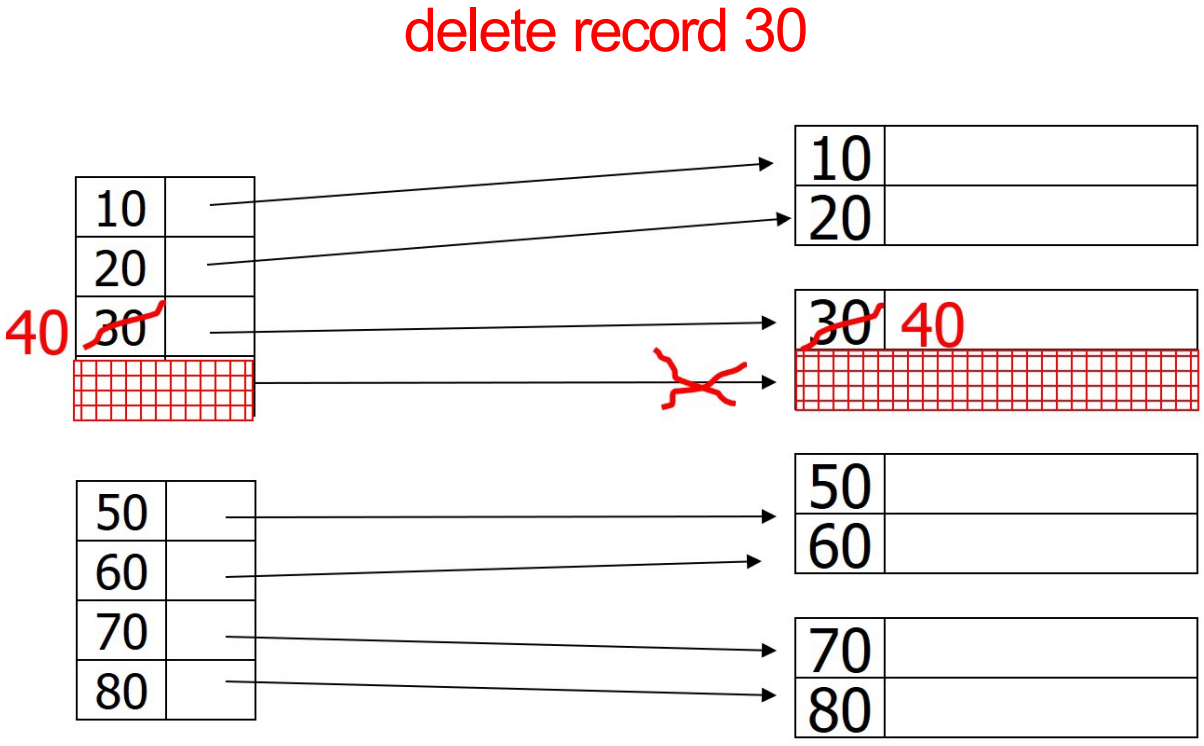
delete record 30



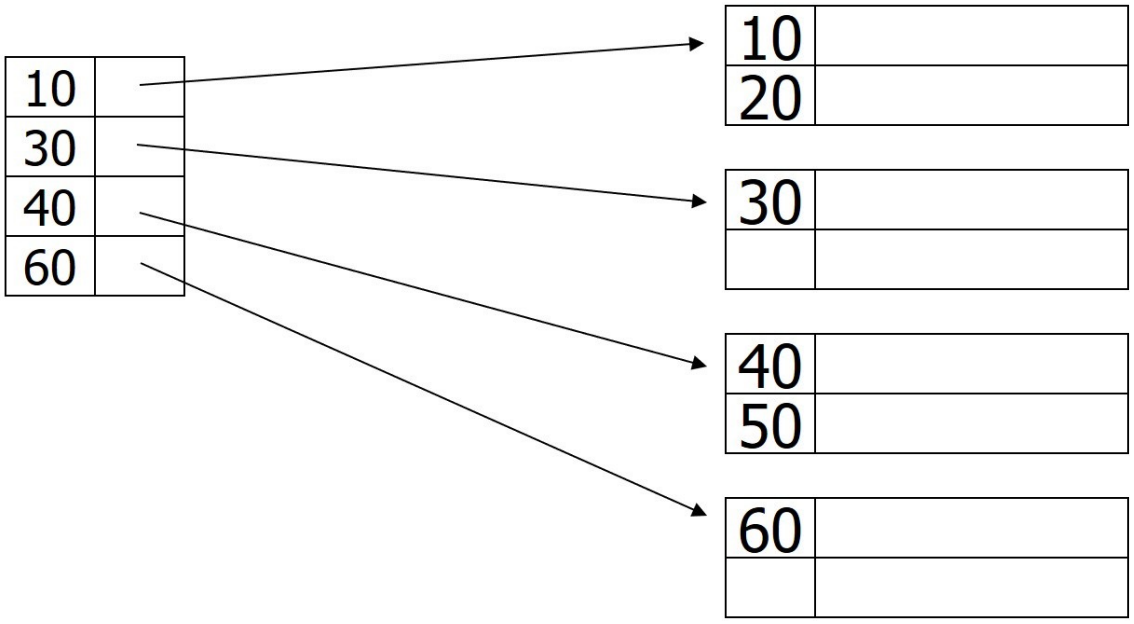
Deletion from dense index



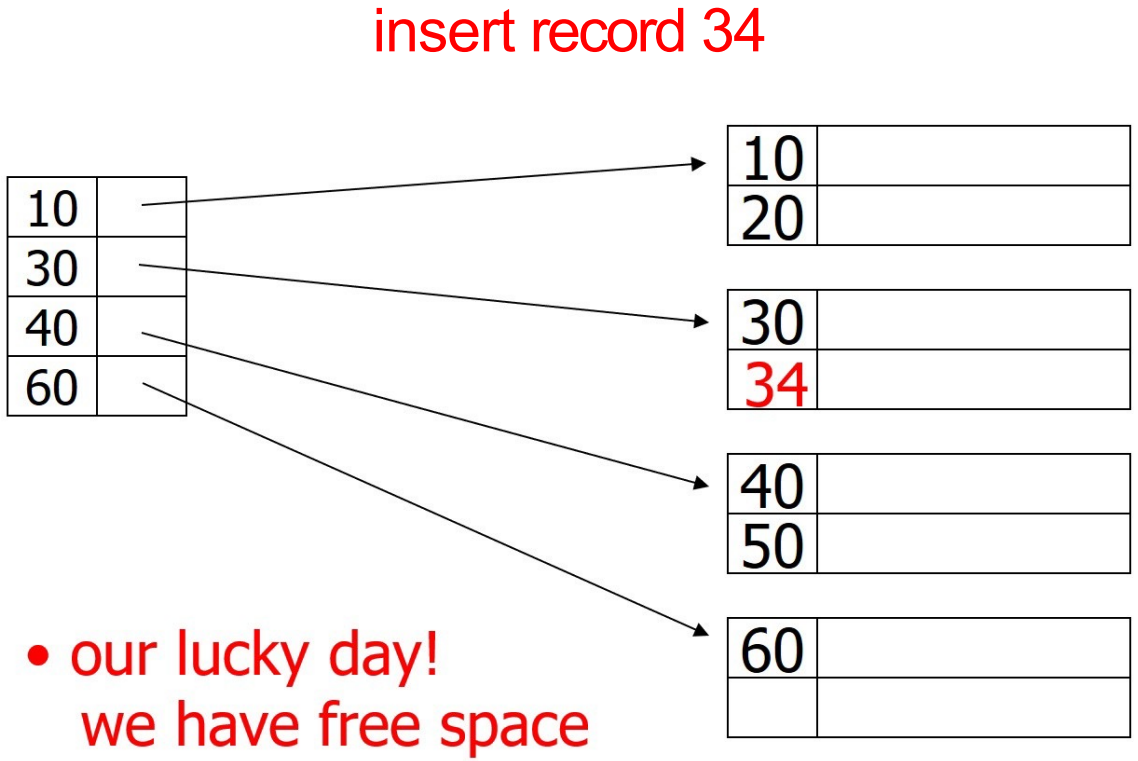
Deletion from dense index



Insertion, sparse index case

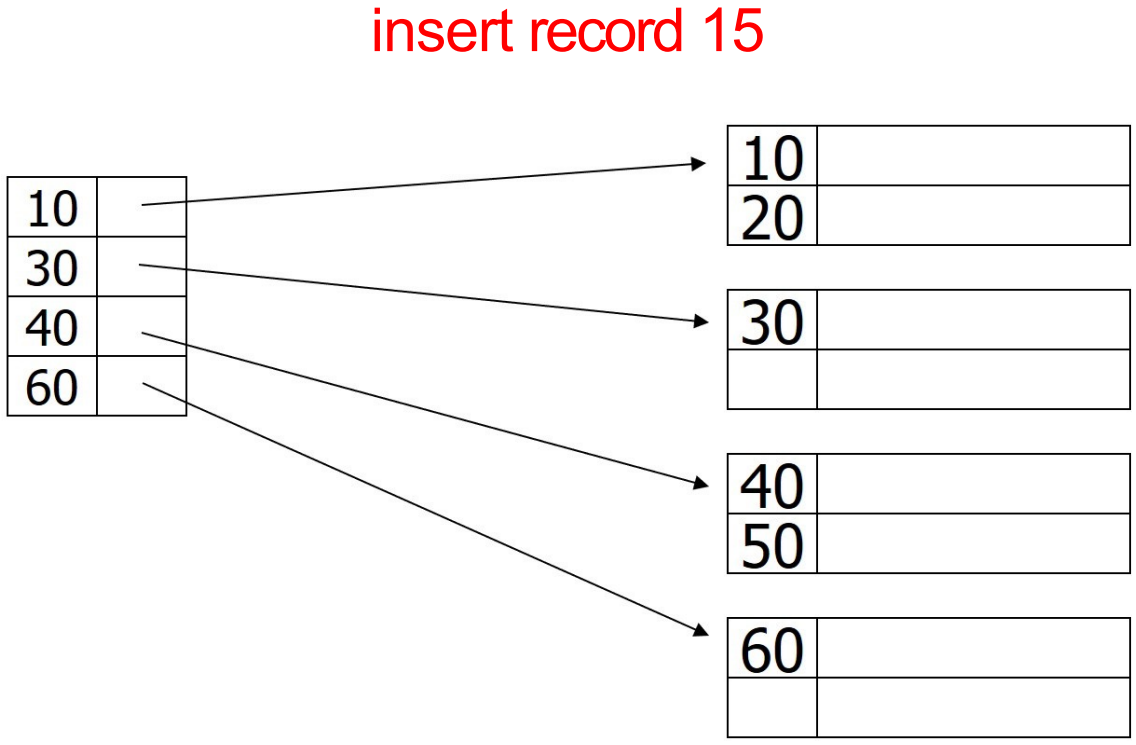


Insertion, sparse index case

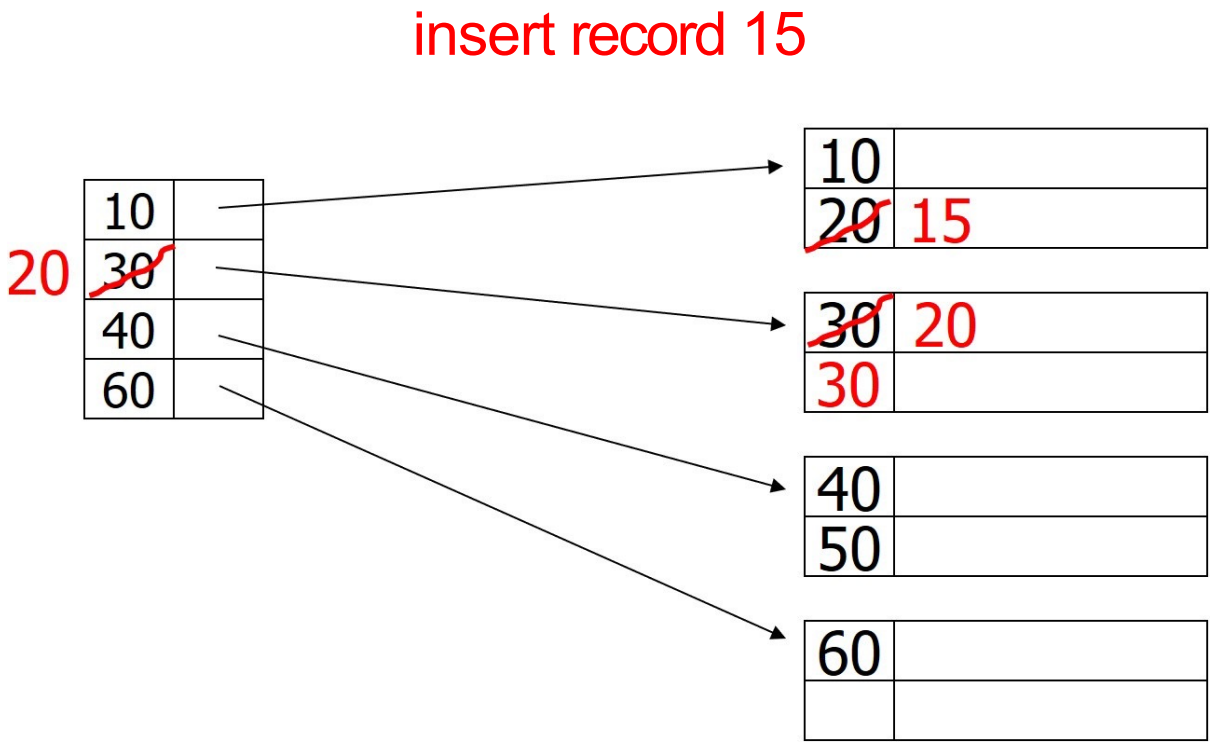


- our lucky day!
  we have free space
  where we need it!

Insertion, sparse index case



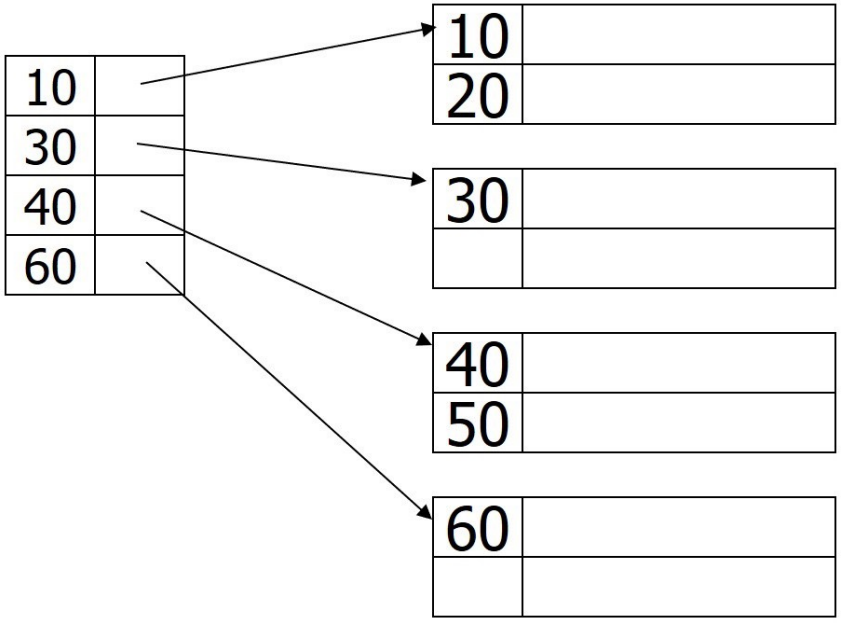
Insertion, sparse index case



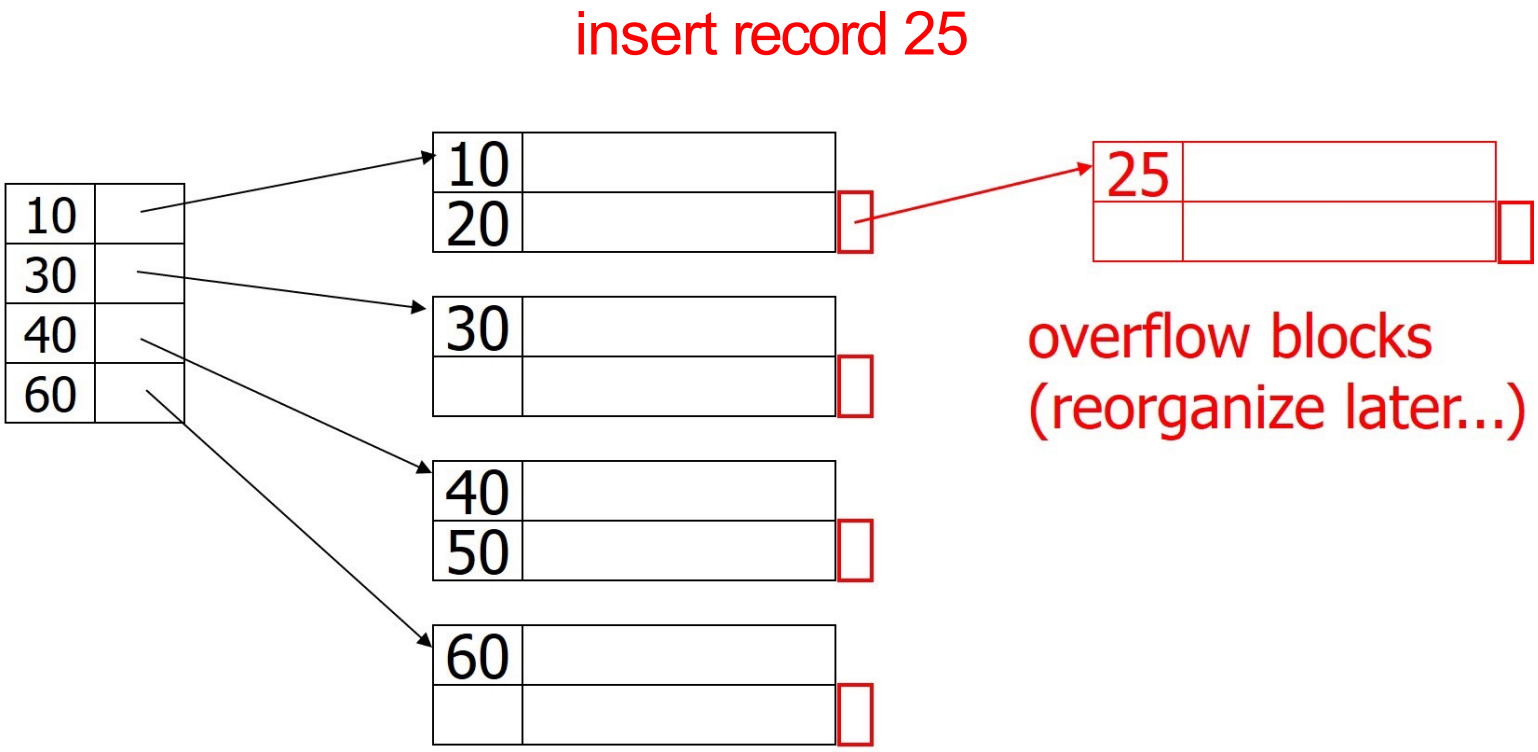
- Illustrated: Immediate reorganization
- Variation
  - insert new block (chained file)
  - update index

Insertion, sparse index case

insert record 25



Insertion, sparse index case



Insertion, dense index case

- Similar
- Often more expensive . . .

Topics

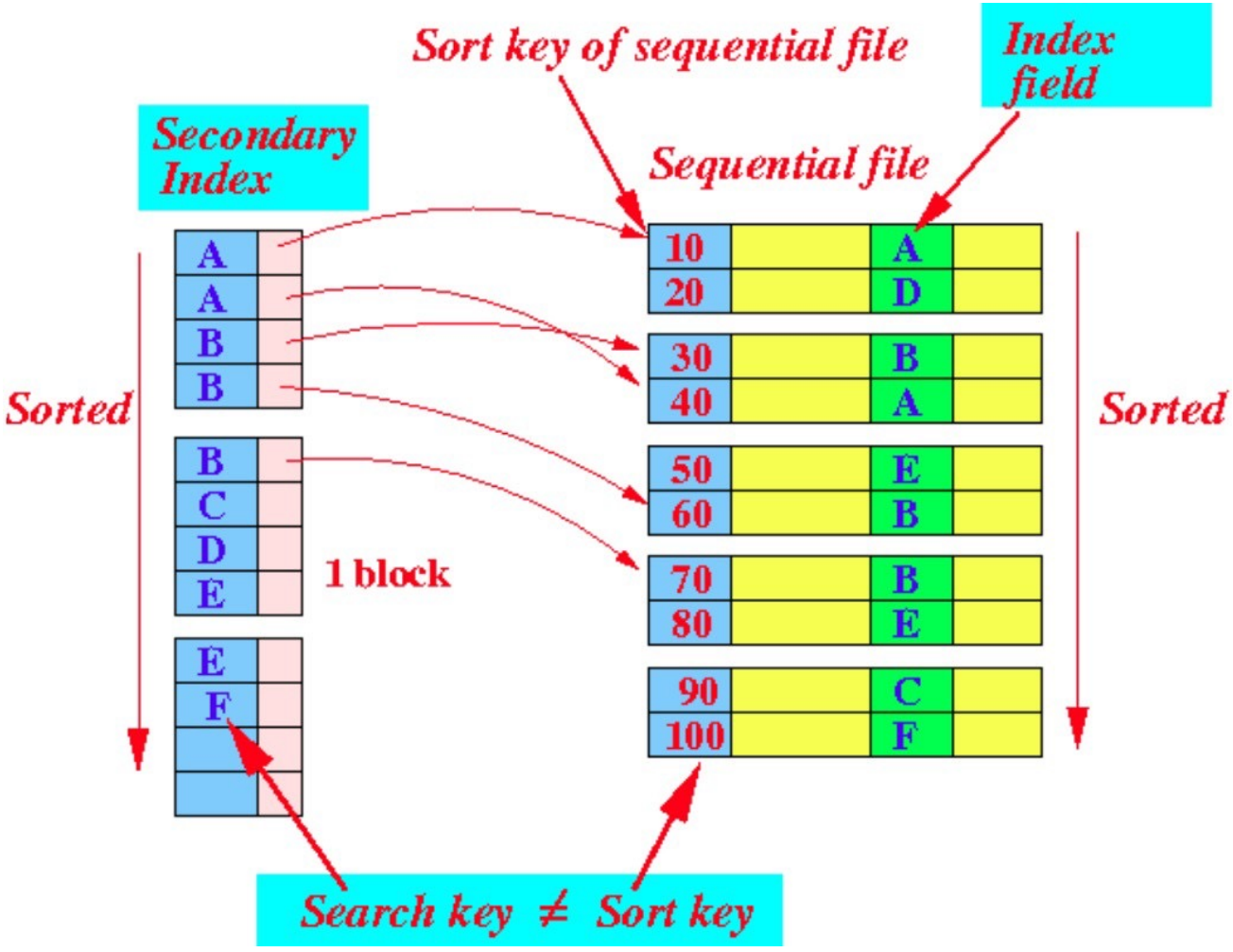
- Conventional indexes
  - Basic Ideas: sparse, dense, multi-level . . .
  - Duplicate Keys
  - Deletion/Insertion
  - Secondary indexes
- B-Trees
- Hashing schemes

Secondary Indexes

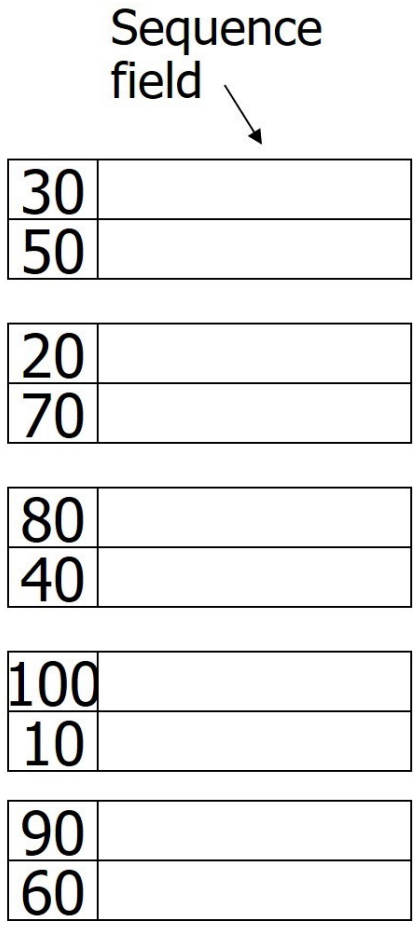
- Sometimes we want multiple indexes on a relation.
  - Ex: search `Candies(name,manf)` both by `name` and by `manufacturer`
- Typically the file would be sorted using the key (ex: `name`) and the `primary index` would be on that field.
- The `secondary index` is on any other attribute (ex: `manf`).
- `Secondary index` also facilitates finding records, but cannot rely on them being sorted

Recall: Secondary index

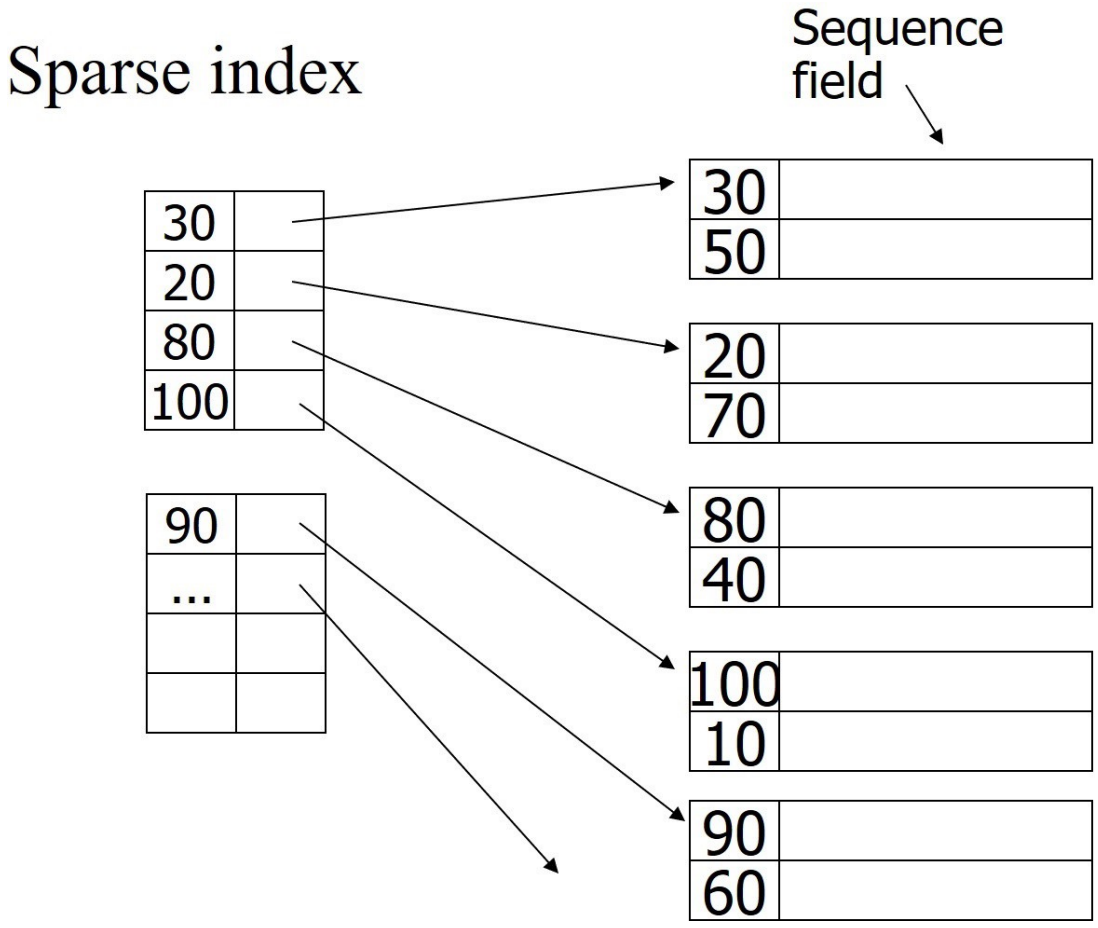
- `Secondary index`: an ordered index whose search key is **NOT** the sort key used for the sequential file



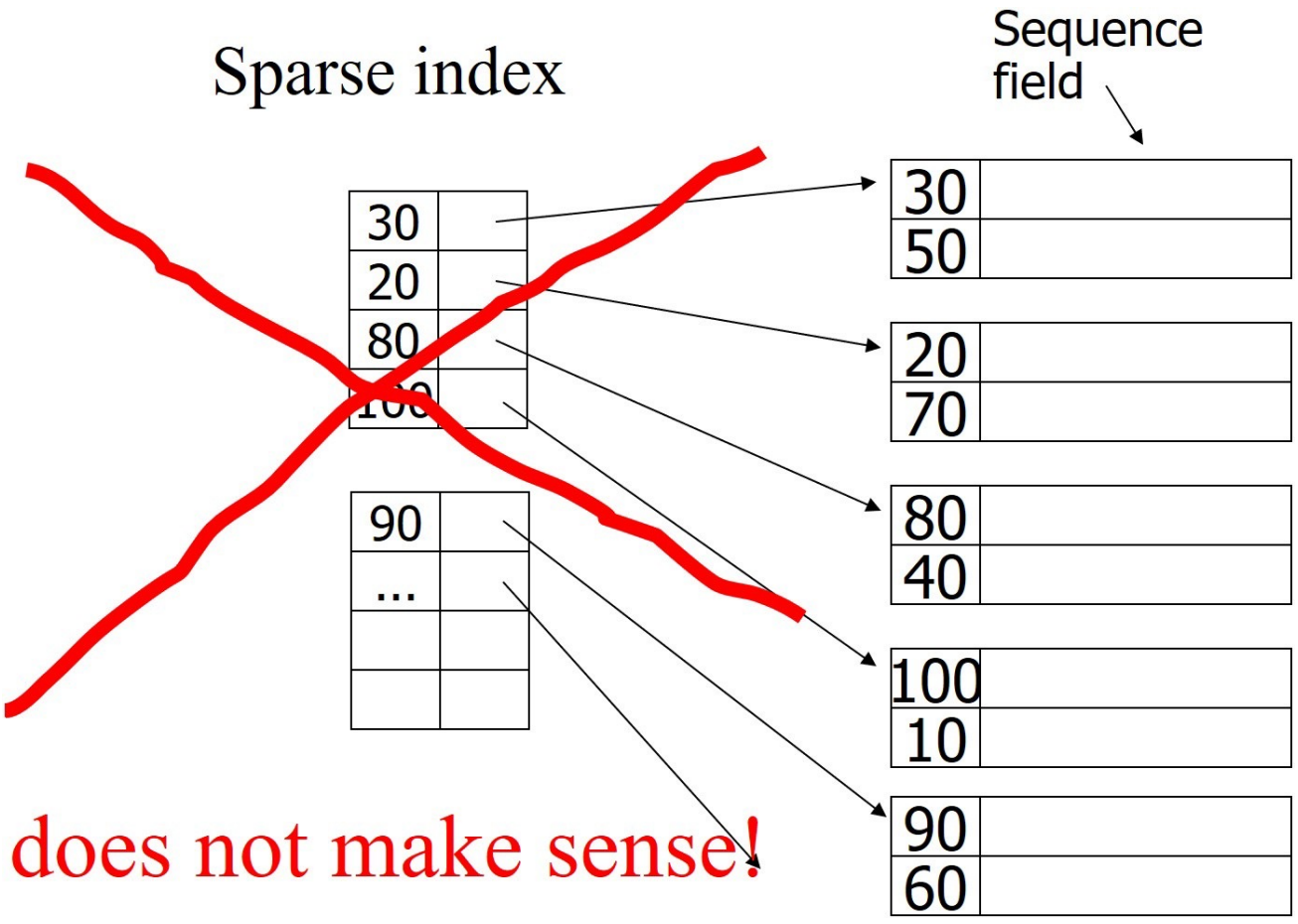
Sparse Secondary Index?



Sparse Secondary Index?



Sparse Secondary Index?



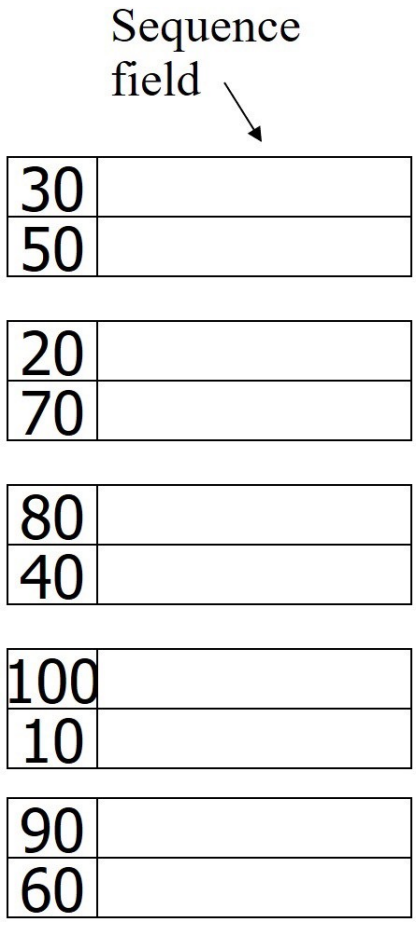
Sparse Secondary Index?

- No!
- Since records are not sorted on that key, cannot predict the location of a record from the location of any other record.
- Thus **secondary indexes** are always **dense**.

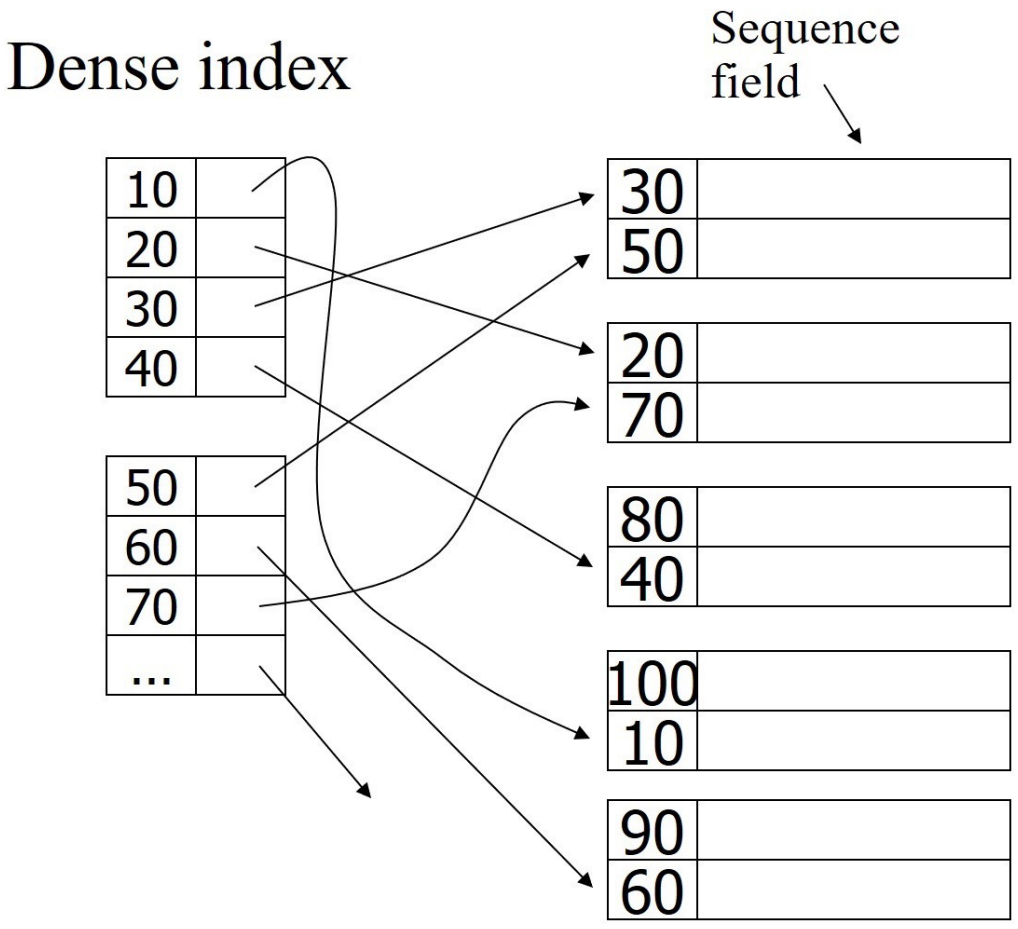
Design of Secondary Indexes

- 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)

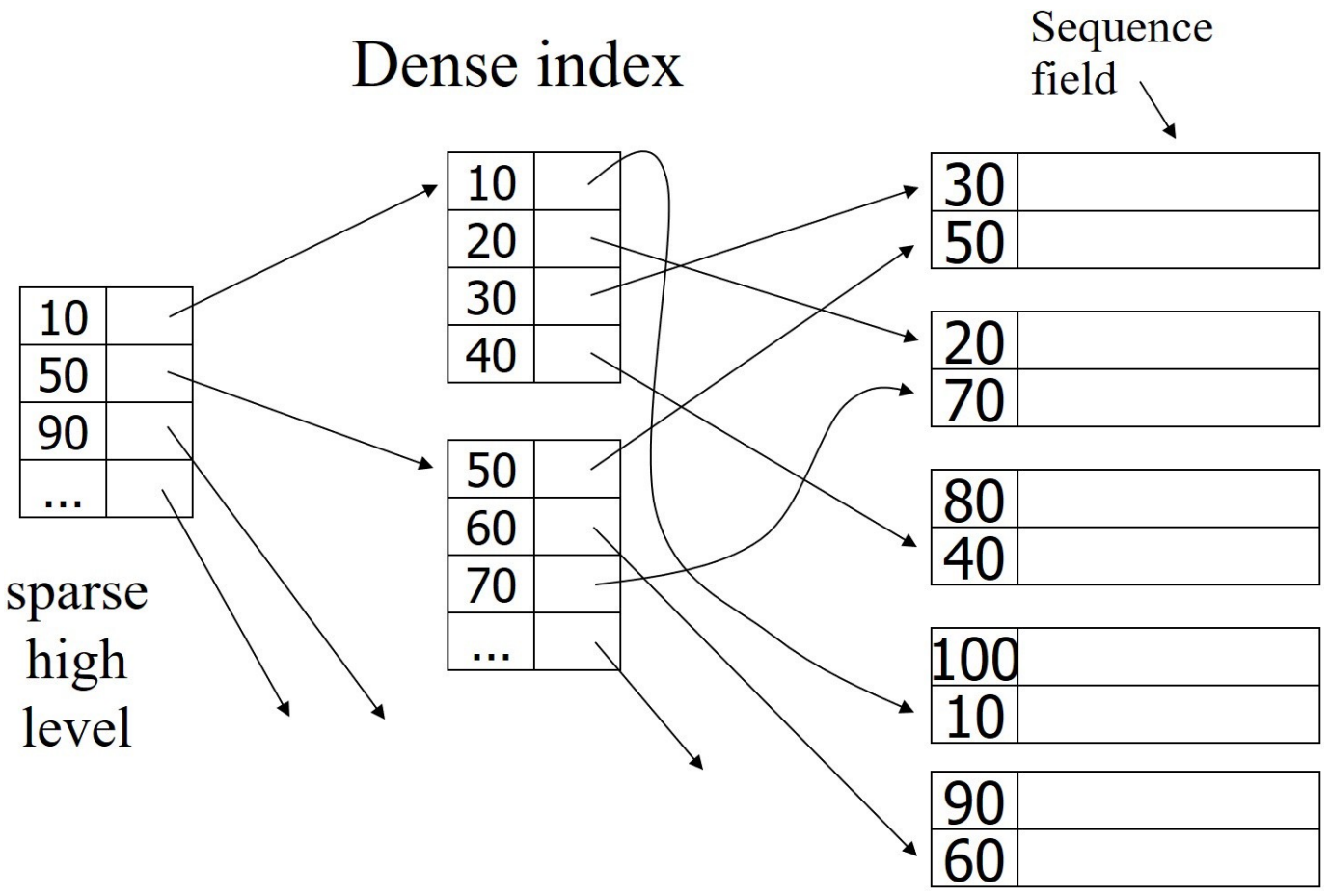
Secondary indexes



Secondary indexes



Secondary indexes



Secondary indexes

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

Secondary Index and Duplicate Keys

- Scheme in previous diagram wastes 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.

Secondary Index and Duplicate Keys

- one option

20	
10	

20	
40	

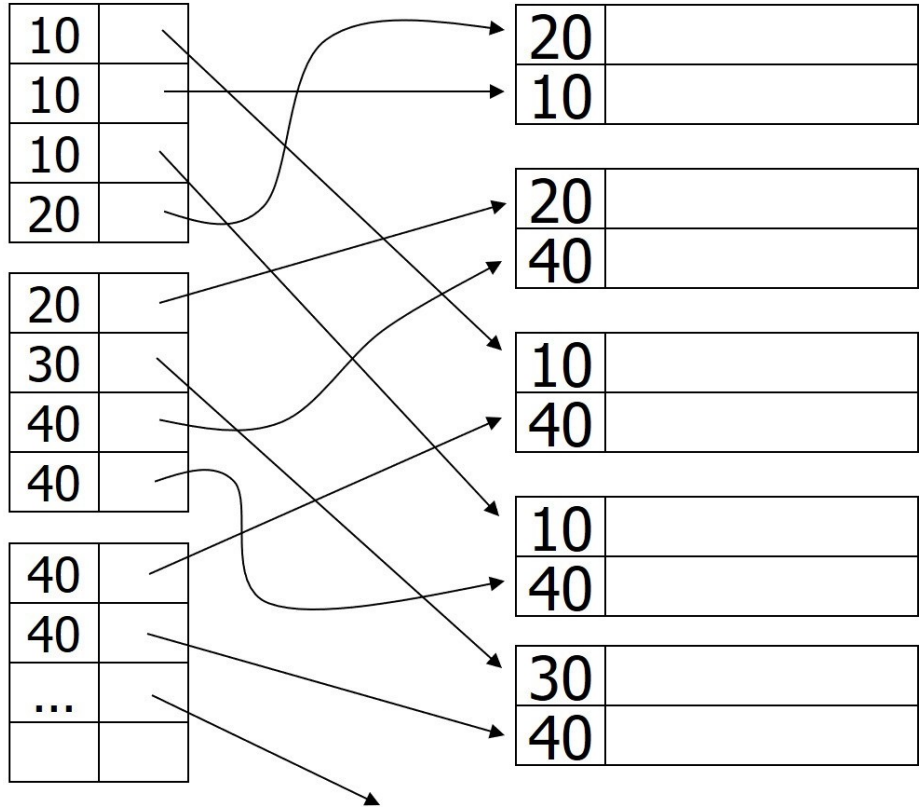
10	
40	

10	
40	

30	
40	

Secondary Index and Duplicate Keys

- one option



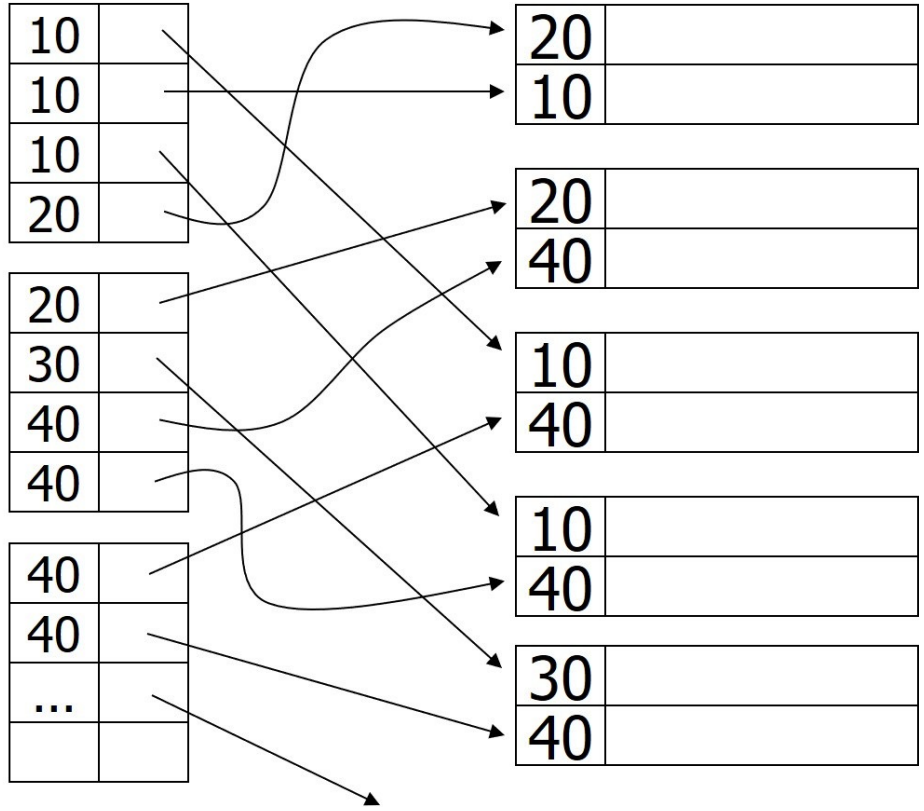
Secondary Index and Duplicate Keys

- one option

Problem:

excess overhead!

- disk space
- search time



Secondary Index and Duplicate Keys

- another option

20	
10	

20	
40	

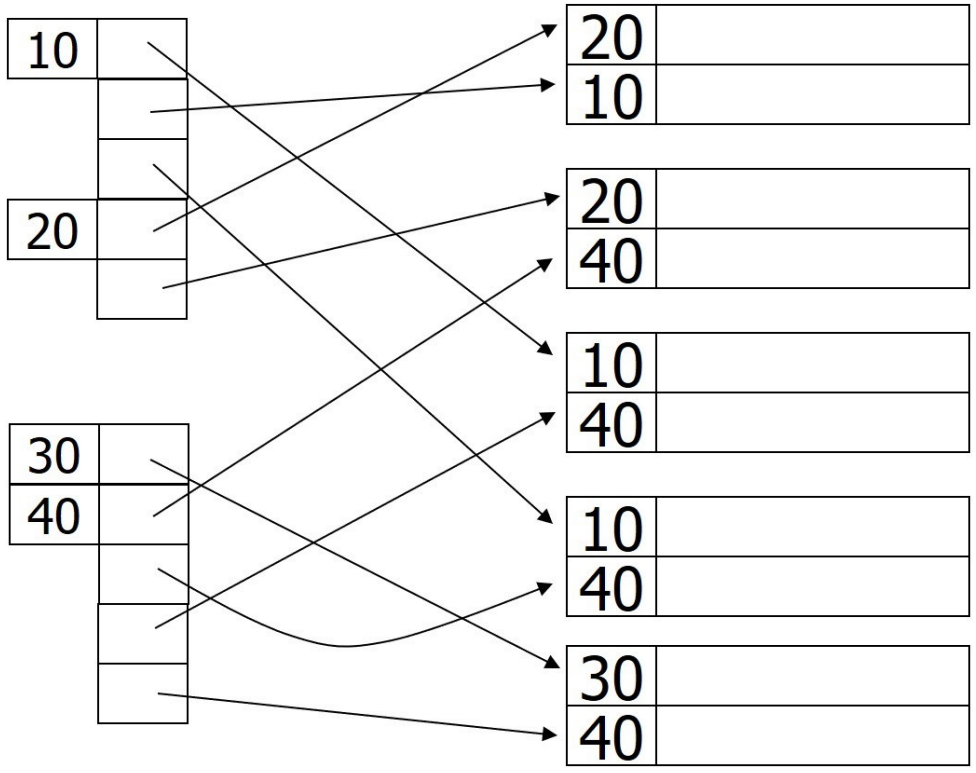
10	
40	

10	
40	

30	
40	

Secondary Index and Duplicate Keys

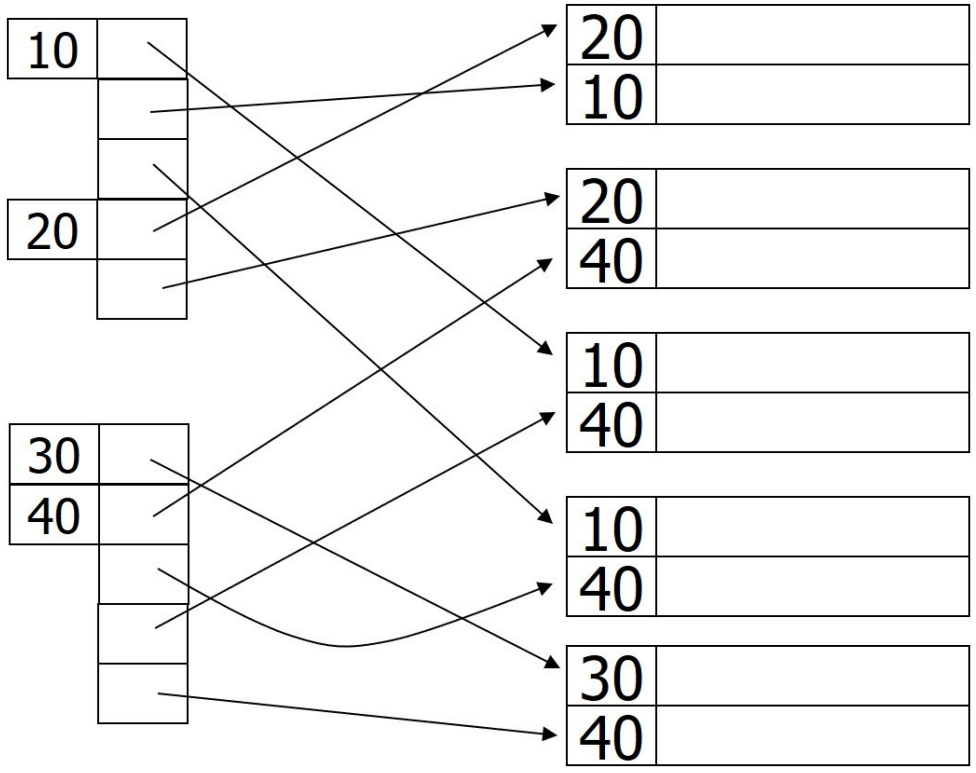
- another option



Secondary Index and Duplicate Keys

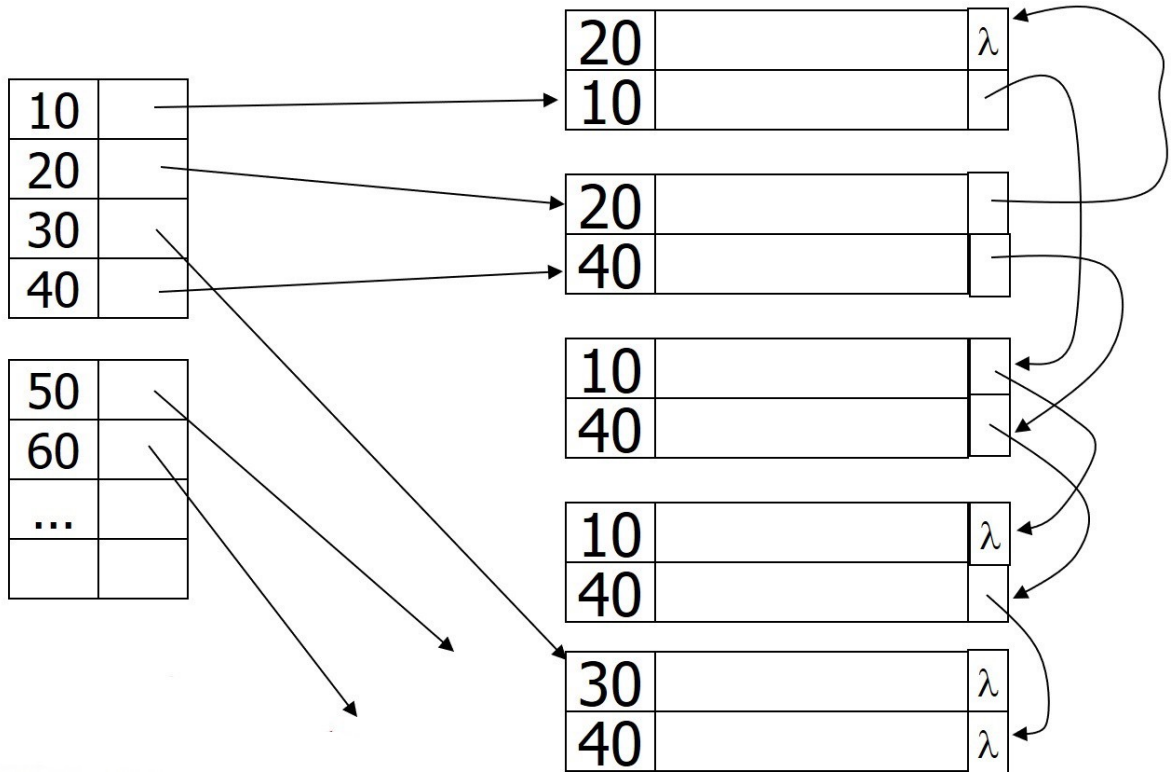
- another option

Problem:
variable size
records in
index!



Secondary Index and Duplicate Keys

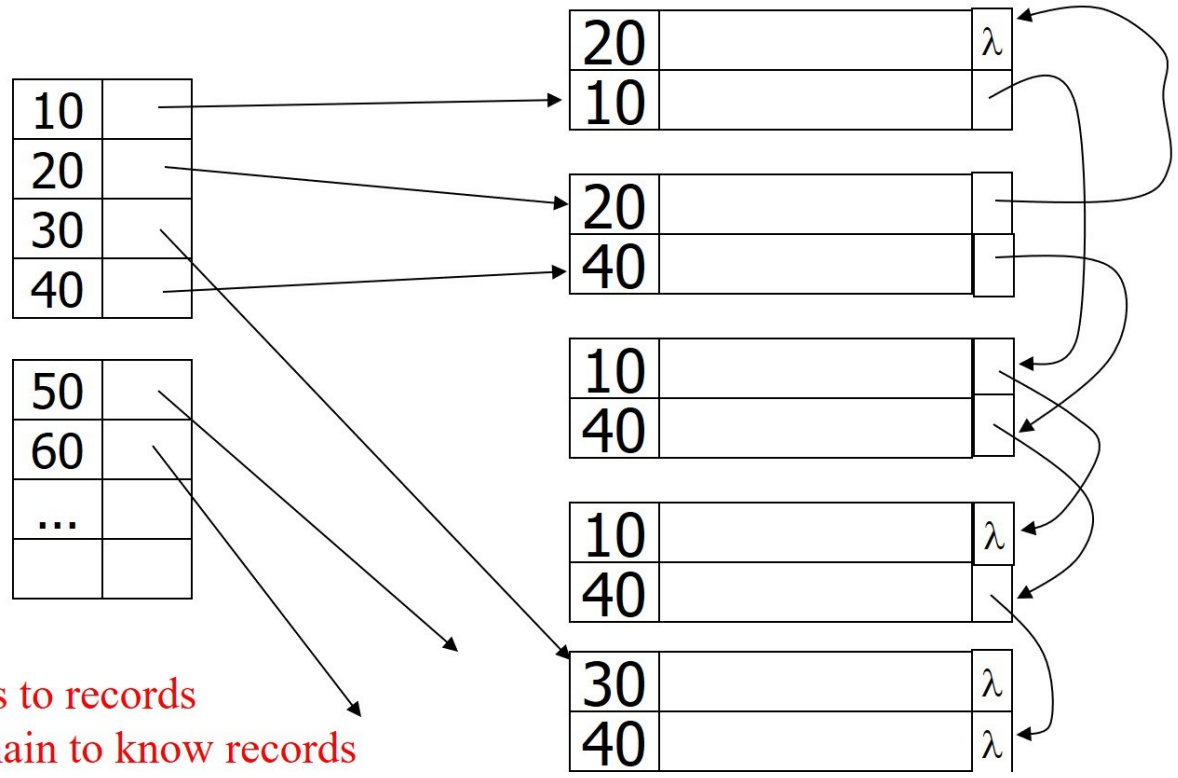
- another idea: Chain records with same key option



lambda indicates the last record in the chain

Secondary Index and Duplicate Keys

- another idea: Chain records with same key option

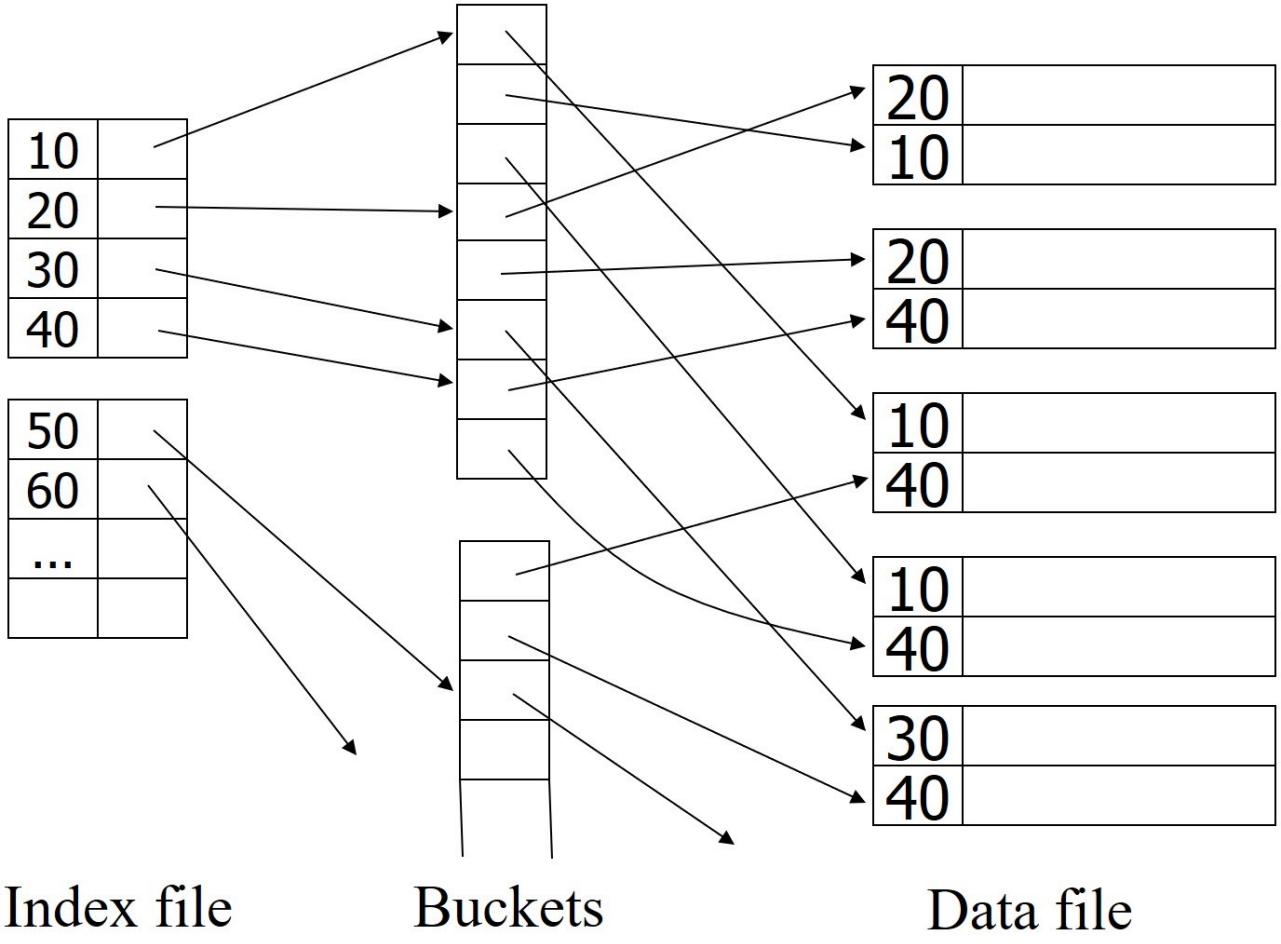


Problems:
- Need to add fields to records
- Need to follow chain to know records

Buckets

- To avoid repeating values, use a level of indirection
- Put **buckets** between the **secondary index** file and the data file
- One entry in index for each `search key` K; its pointer goes to a location in a "bucket file", called the "bucket" for K
- **Bucket** holds pointers to all records with `search key` K

Secondary Index and Duplicate Keys



Why “bucket” idea is useful

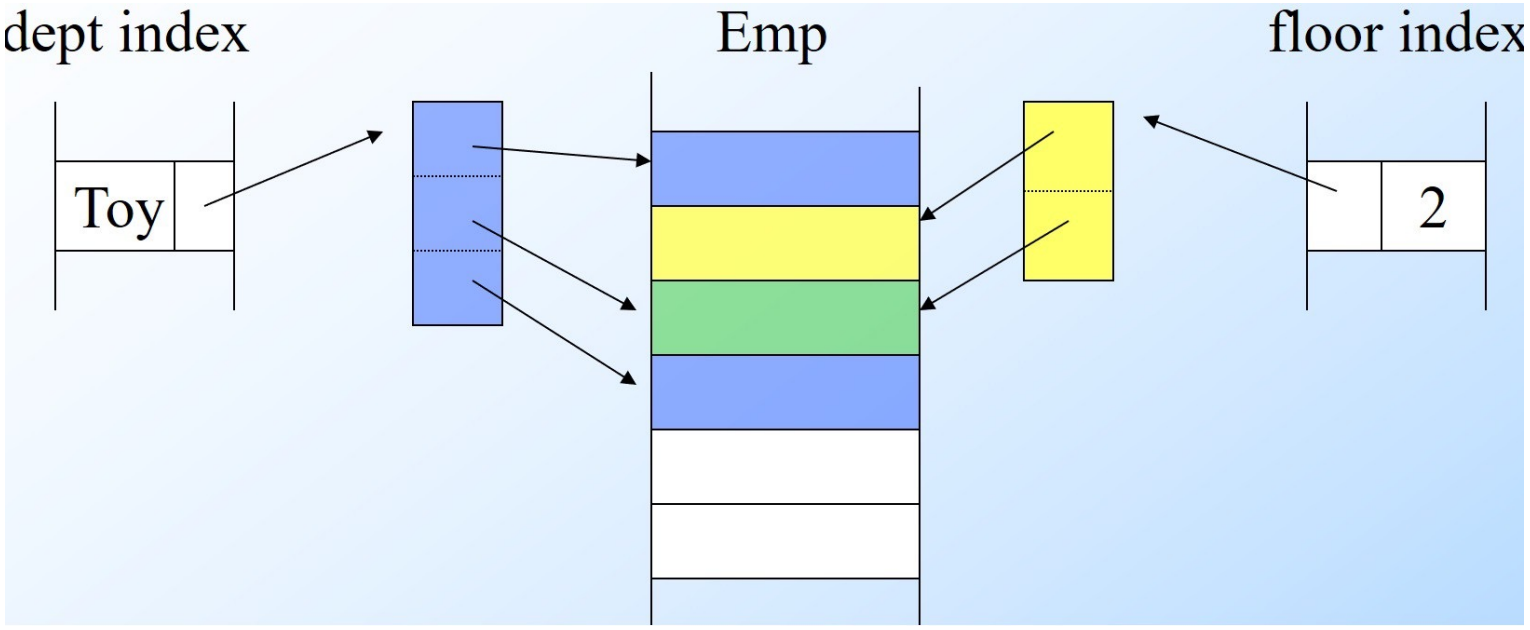
- Saves space as long as search-keys are larger than pointers and average key appears at least twice
- We can use the pointers in the buckets to help answer queries without ever looking at most of the records in the data file.
  - When there are several conditions to a query, and each condition has a 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:

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

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

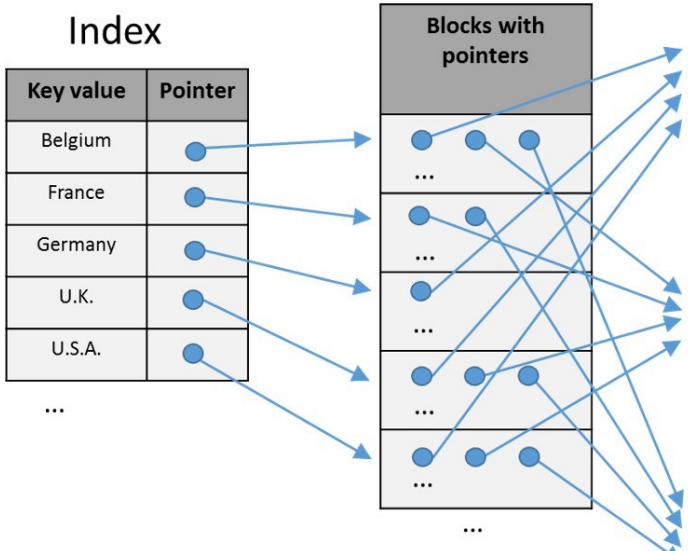


- Intersect `Toy dept` bucket and `floor 2` bucket to get set of matching **Emp**'s
- Retrieving the minimum possible number of data blocks. Saves disk I/O's

Inverted files

- Inverted file defines index over non-unique, non-ordering search key of data set
- Index entries: <key value, block address>
- Block address refers to block containing record pointers or block pointers to all records with that particular key value
- Requires additional random block access to block with pointers to records
- Queries that involve multiple attribute types can be executed efficiently by taking the intersection of blocks with pointers

Inverted files



File with stored records

CustomerID	FirstName	LastName	Country	Year of birth	Gender
10023	Bart	Baesens	Belgium	1975	M
10098	Charlotte	Bobson	U.S.A.	1968	F
10233	Donald	McDonald	U.K.	1960	M
10299	Heiner	Pilzner	Germany	1973	M
10351	Simonne	Toutdroit	France	1981	F
10359	Seppe	Vanden Broucke	Belgium	1989	M
10544	Bridget	Charlton	U.K.	1992	F
11213	Angela	Kissinger	U.S.A.	1969	F
11349	Henry	Dumortier	France	1987	M
11821	Wilfried	Lemahieu	Belgium	1970	M
12111	Tim	Pope	U.K.	1956	M
12194	Naomi	Leary	U.S.A.	1999	F

...

This idea used in text information retrieval

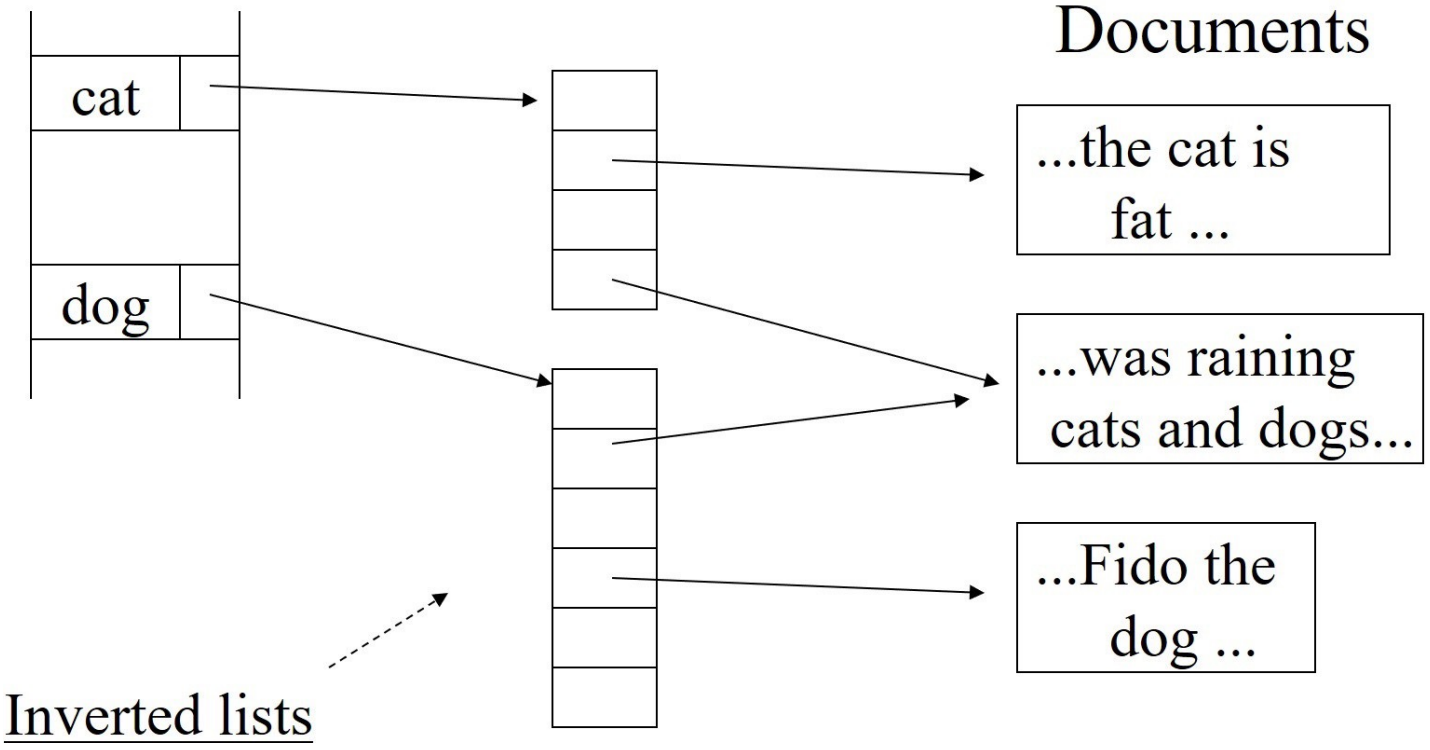
Documents

...the cat is fat ...

...was raining cats and dogs...

...Fido the dog ...

This idea used in text information retrieval



`inverted index` consists of a set of word-pointer pairs; the words are in effect the search key for the index.

IR Queries

- Find articles with “cat” and “dog”
- Find articles with “cat” or “dog”
- Find articles with “cat” and not “dog”
- Find articles with “cat” in title
- Find articles with “cat” and “dog” within 5 words

Summary so far

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

Conventional indexes

- Advantage
  - Simple
  - Index is sequential file, good for scans
- Disadvantage
  - Inserts expensive, and/or
  - Lose sequentiality & balance
- Instead use `B+-Tree` data structure to implement index