

CS525: Advanced Database Organization

Notes 4: Indexing and Hashing

Part I: Conventional indexes

Yousef M. Elmehdwi

Department of Computer Science

Illinois Institute of Technology

yelmehdwi@iit.edu

September 4th, 2018

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](#)

Annoncement

- TA: Ziyu Liu, zliu102@hawk.iit.edu

Record Organization¹

- Blocking factor (BF) indicates how many records are stored in single disk block
- For a file with fixed length records, BF is calculated as: $BF = \lfloor \frac{BS}{RS} \rfloor$, where BS denotes the block size and RS is the record size, both represented in bytes
- For variable length records, BF denotes the average number of records in a block
- Blocking factor determines how many records are retrieved with a single read operation

¹Slide credit: Principles of Database Management <http://www.pdbmbook.com>

This Chapter

- How to find a record quickly, given a key

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

Introductory Concepts

- **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

Introductory Concepts

- **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 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: $\frac{B}{2}$ sequential block accesses
 - binary search: $\log_2 B$ random block accesses

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:

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

- 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

Indexed Sequential File Organization

- File is divided into intervals or partitions
- Each interval is represented by index entry containing search key value of first record in interval and pointer to physical position of first record in interval
- Pointer can be block pointer (referring to physical block address) or record pointer (consisting of combination of block address and record id or offset within block)
- Index is sequential file, ordered according to search key values with entries: <search key value, block pointer or record pointer>
- Search key can be atomic (e.g., a CustomerID) or composite (e.g. Year of Birth and Gender)

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)

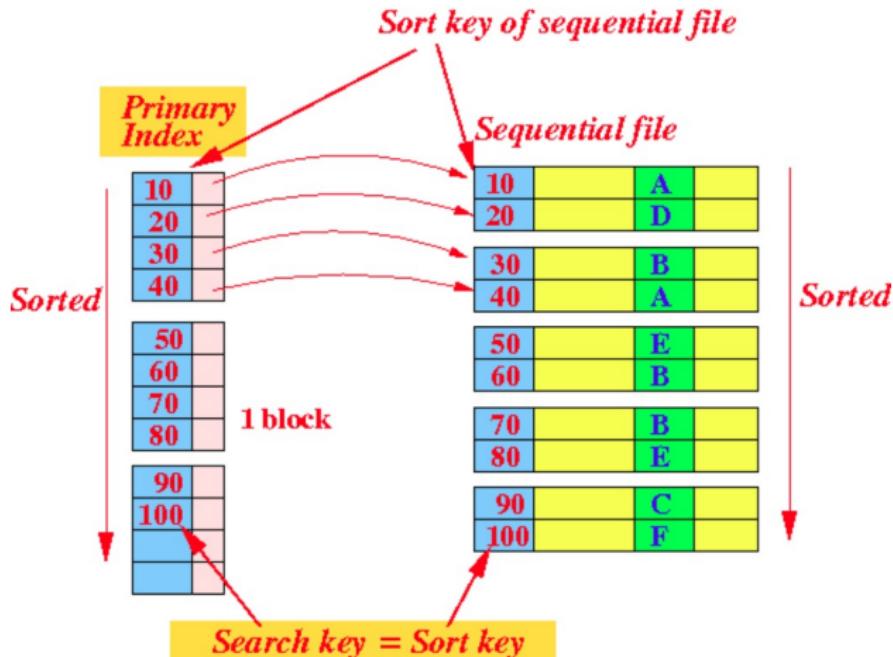
Index: Ordered Index

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



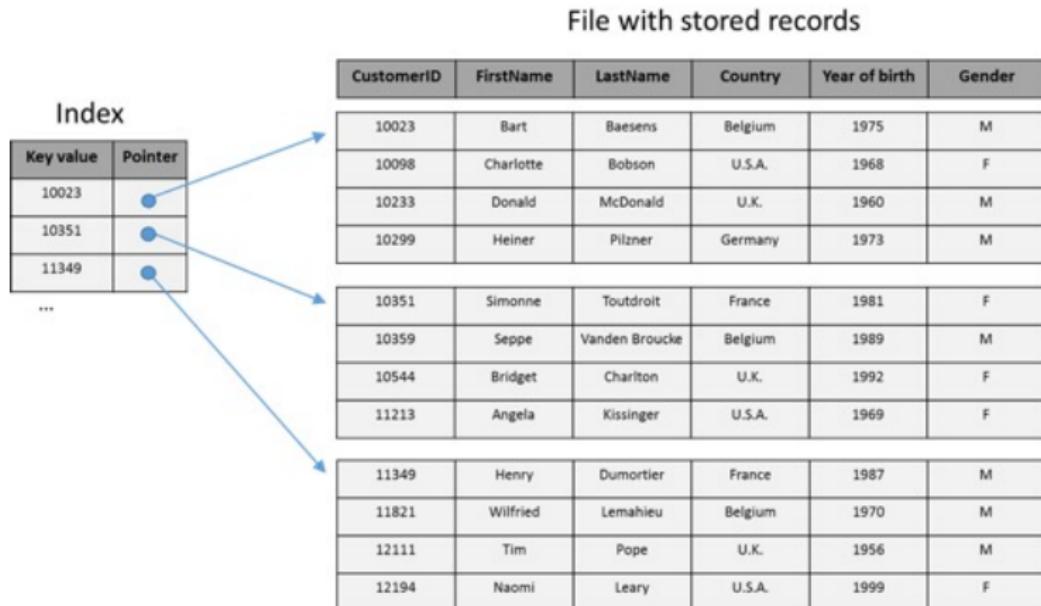
Index: Primary index

- 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



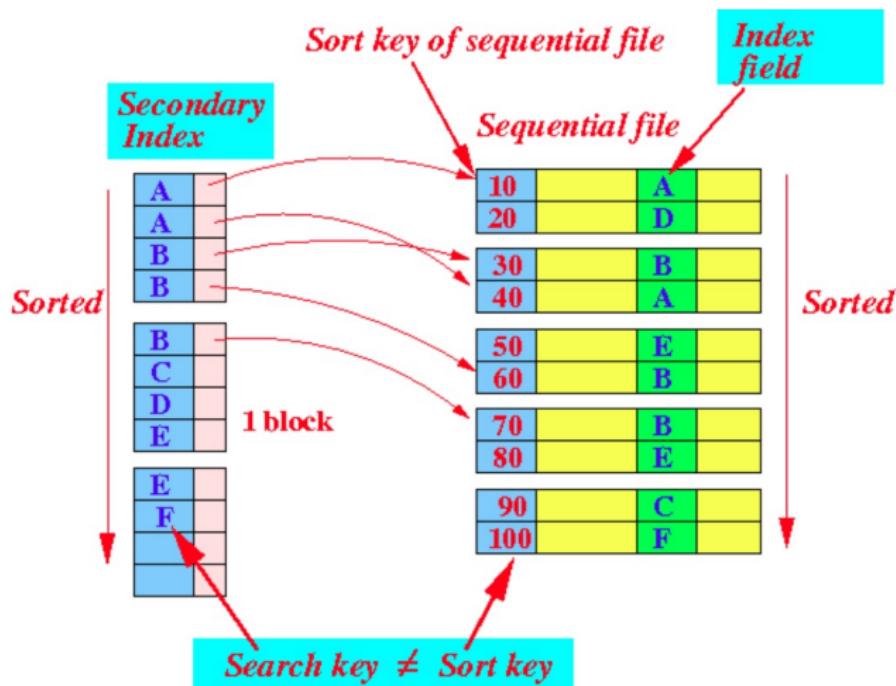
Index: Primary index: Example

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



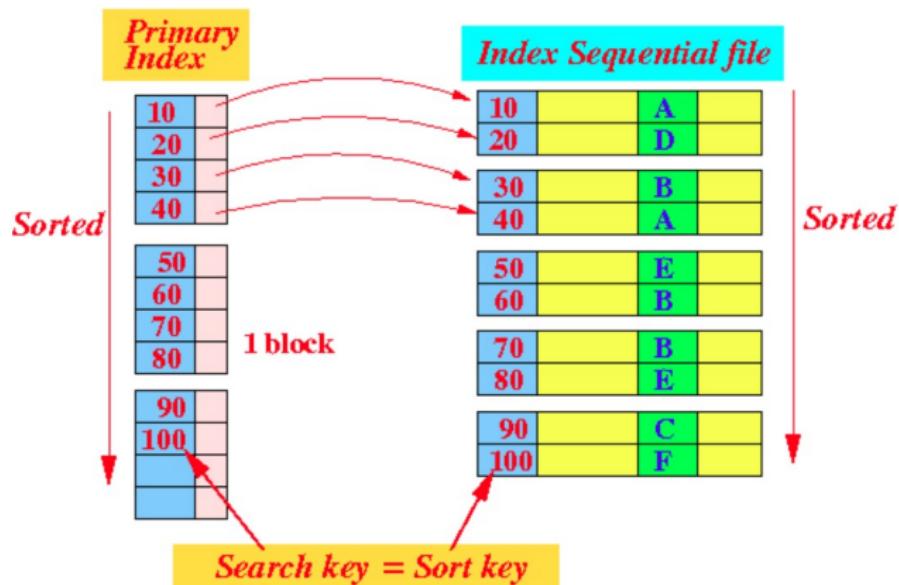
Index: Secondary index

- 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

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

Sequential File

10	
20	

30	
40	

50	
60	

70	
80	

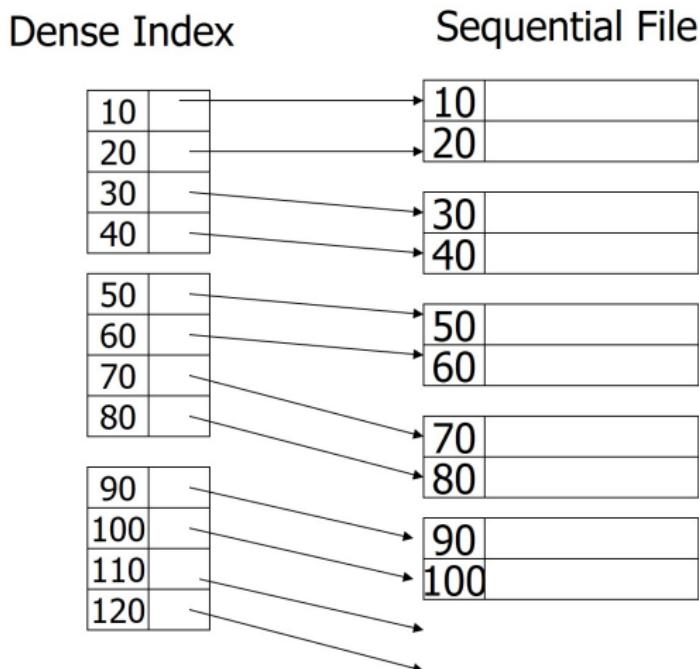
90	
100	

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.*



- Secondary index is always a dense index. **Why?** search key ≠ Sort

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.

Sparse Index

The diagram illustrates a mapping between two tables. The source table on the left has four rows with values 10, 30, 50, and 70. The target table on the right also has four rows. Arrows point from the first three rows of the source to the first three rows of the target. Specifically, the first row of the source (10) maps to the first row of the target (10), the second row (30) maps to the second row (20), and the third row (50) maps to the third row (30). The fourth row of the source (70) has no arrow pointing to it in the target table.

10	
30	
50	
70	

→

10	
20	
30	

Sequential File

90	
110	
130	
150	
170	
190	
210	
230	

```
graph LR; A[90, 110, 130, 150] --> B[50, 60]; A --> C[70, 80]; A --> D[90, 100];
```

50	
60	
70	
80	
90	
100	

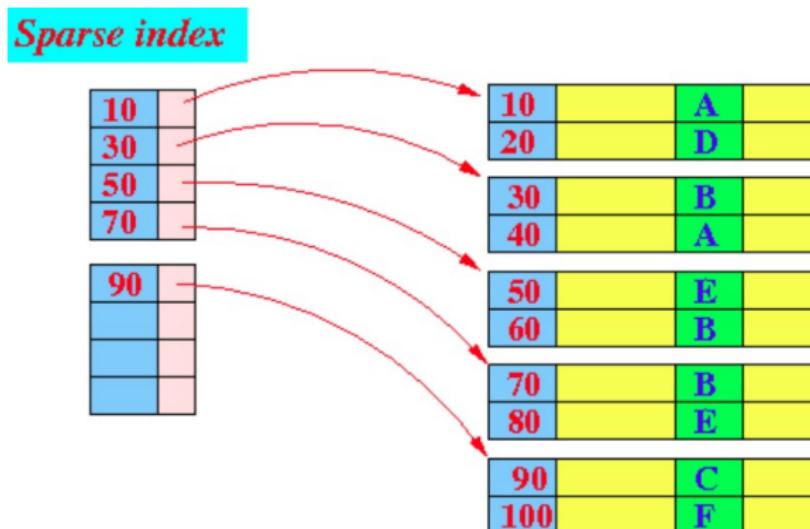
Only first Record
per block in Index.

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

Sparse 2nd level

10	
90	
170	
250	

330	
410	
490	
570	

10	
30	
50	
70	

90	
110	
130	
150	

170	
190	
210	
230	

Sequential File

10	
20	

30	
40	

50	
60	

70	
80	

90	
100	

Comment

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

Question

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

Sparse 2nd level

10	
90	
170	
250	

330	
410	
490	
570	

10	
30	
50	
70	

90	
110	
130	
150	

170	
190	
210	
230	

Sequential File

10	
20	

30	
40	

50	
60	

70	
80	

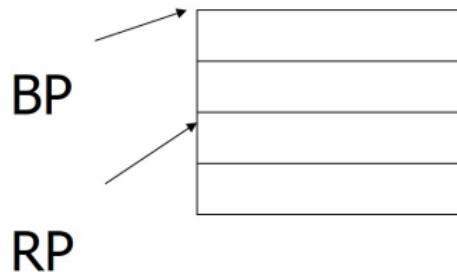
90	
100	

Question

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

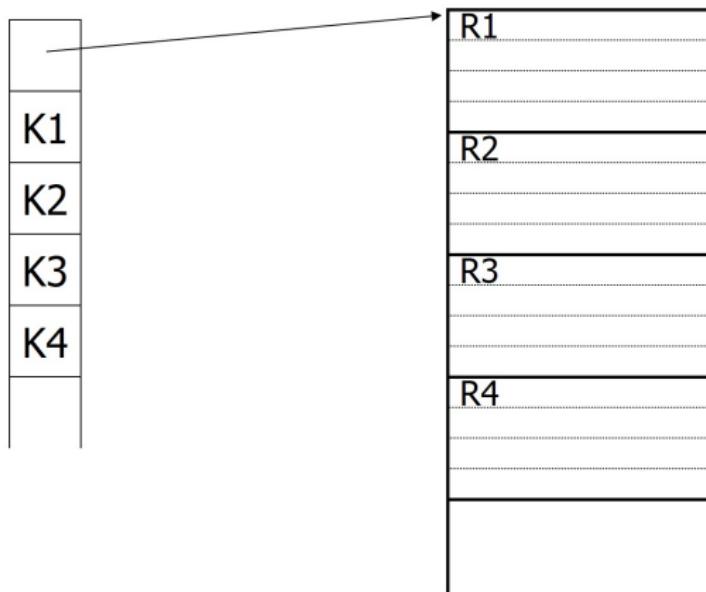
Notes on pointers

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

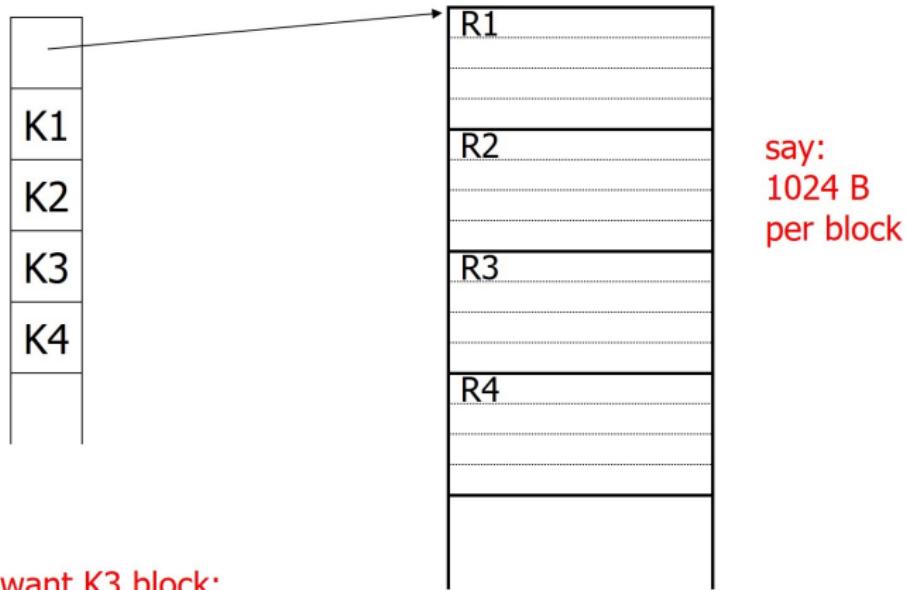


- 2) If file is contiguous, then we can omit pointers (i.e., compute them)

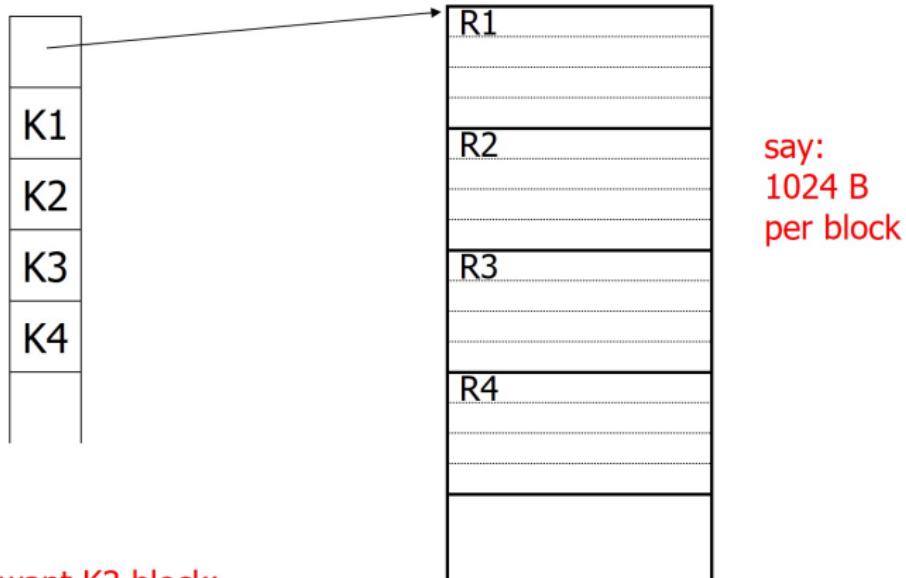
Notes on pointers



Notes on pointers



Notes on pointers



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

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)

- 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	

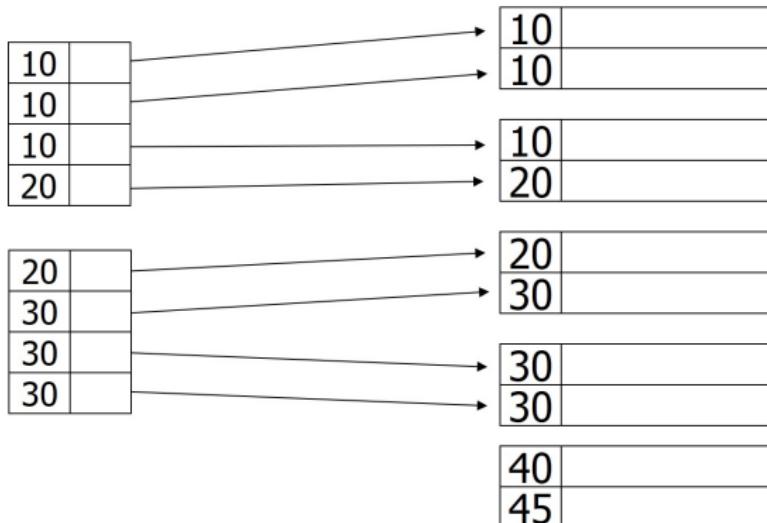
20	
30	

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	

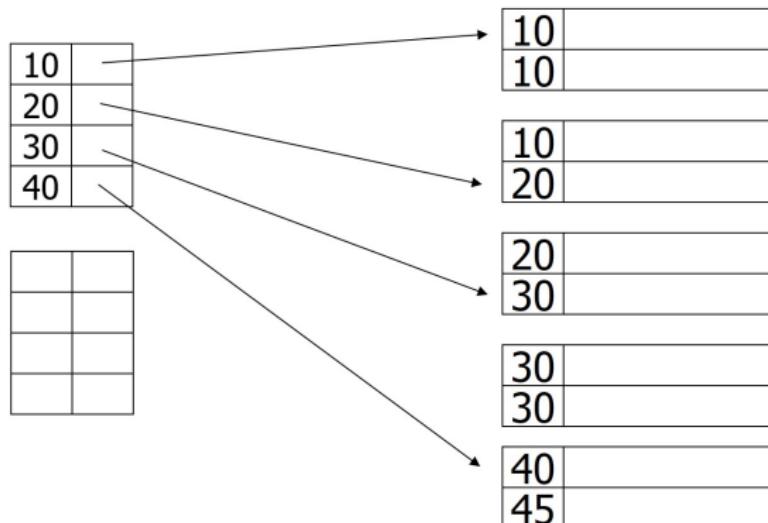
20	
30	

30	
30	

40	
45	

Duplicate Search Keys with Dense Index keys

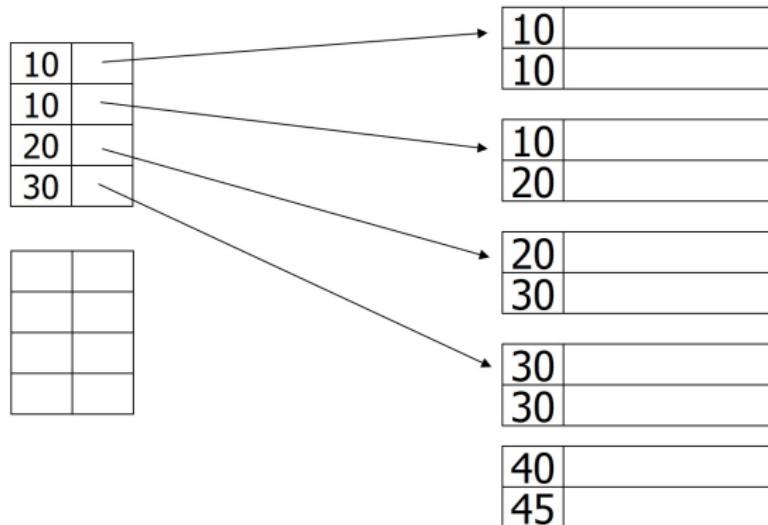
- Dense index, better way? Point to each distinct value!
 - only keep record in index for first data record with each search key value (saves some space in the index)



- 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.



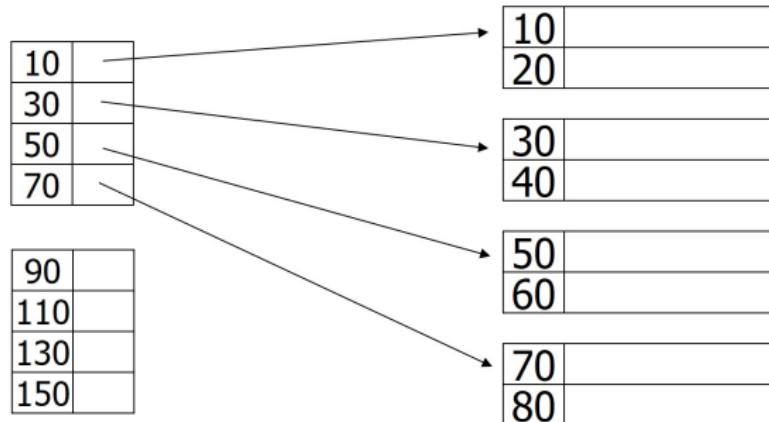
- Find all records with search key 20? Search key =10?

Duplicate search keys with sparse index

- To find all data records with search key K :
 - find last entry (E_1) in index with $\text{key} \leq K$
 - move towards front of index until either reaching entry (E_2) with $\text{key} < K$ 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

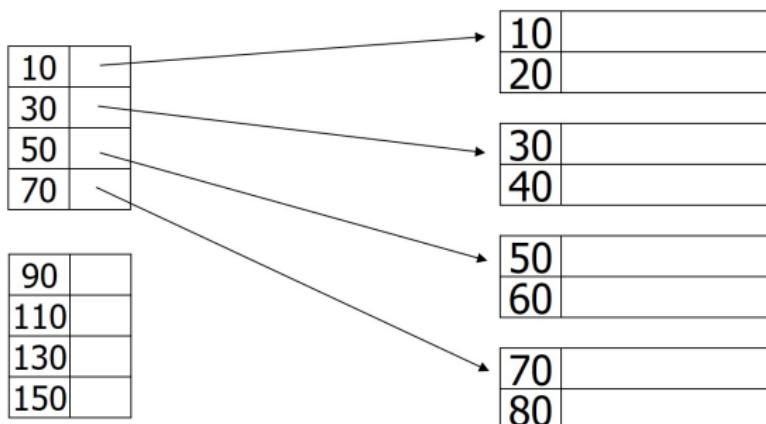
- Duplicate keys
- Deletion/Insertion
- Secondary indexes

Deletion from sparse index



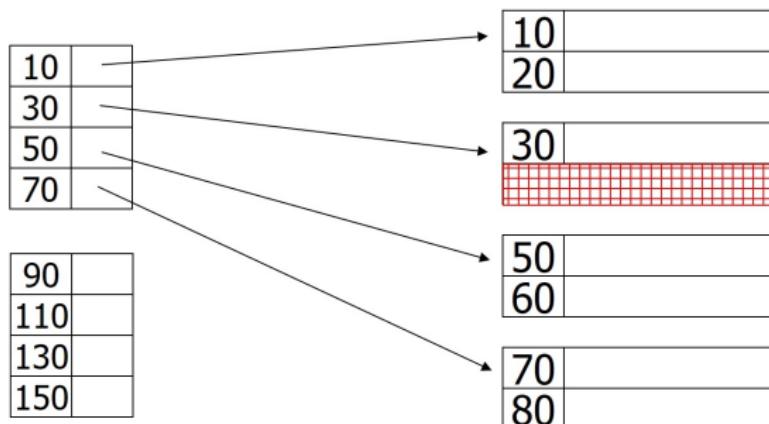
Deletion from sparse index

delete record 40



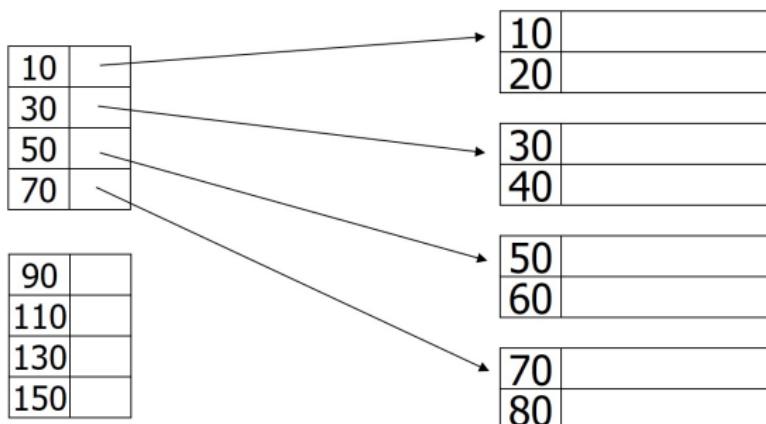
Deletion from sparse index

delete record 40



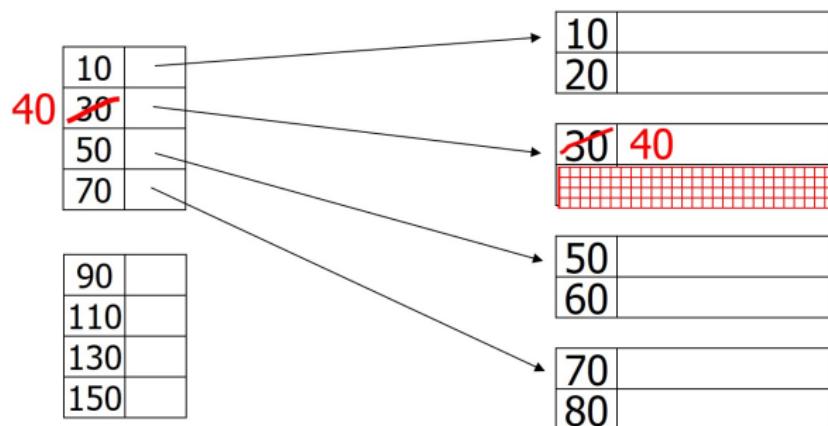
Deletion from sparse index

delete record 30



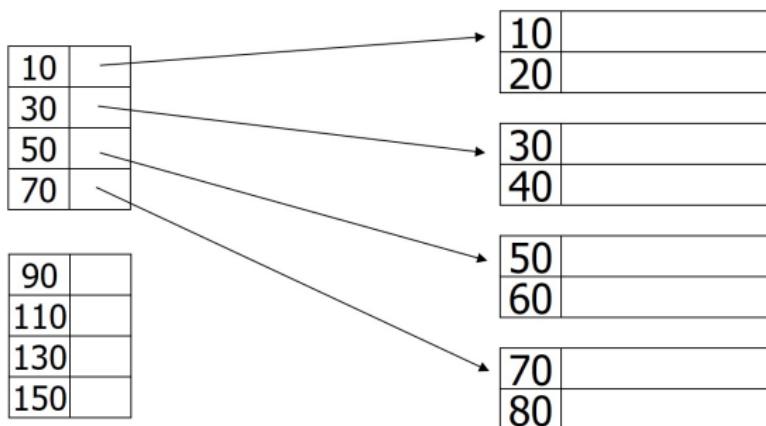
Deletion from sparse index

delete record 30



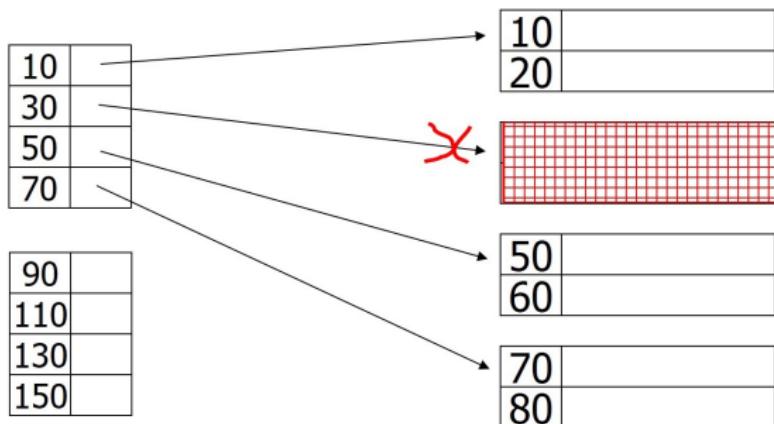
Deletion from sparse index

delete records 30 & 40



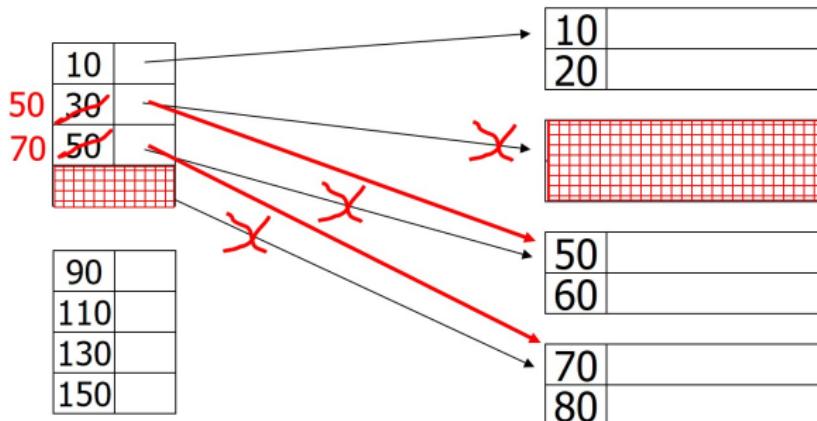
Deletion from sparse index

delete records 30 & 40

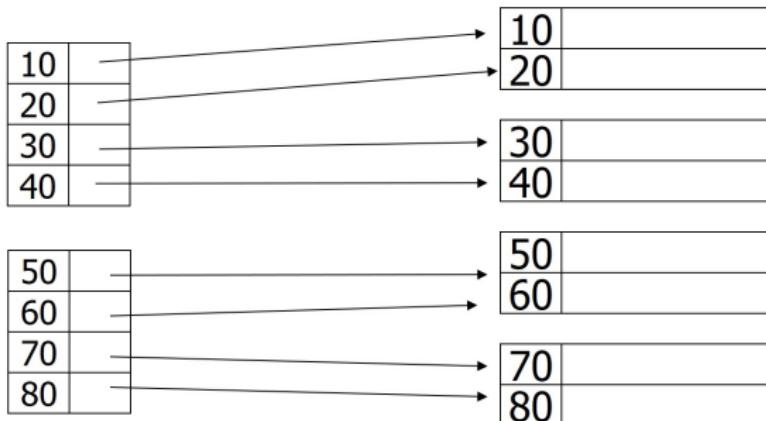


Deletion from sparse index

delete records 30 & 40

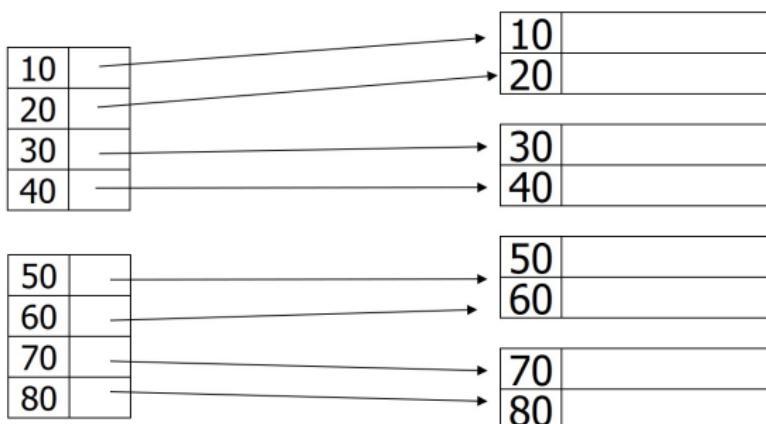


Deletion from dense index



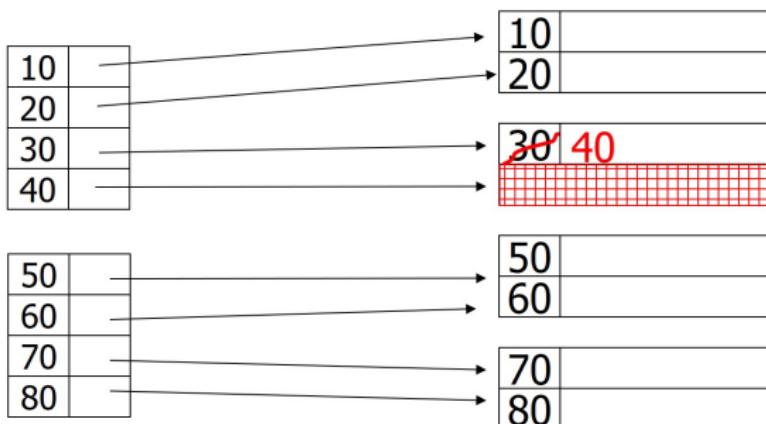
Deletion from dense index

delete record 30



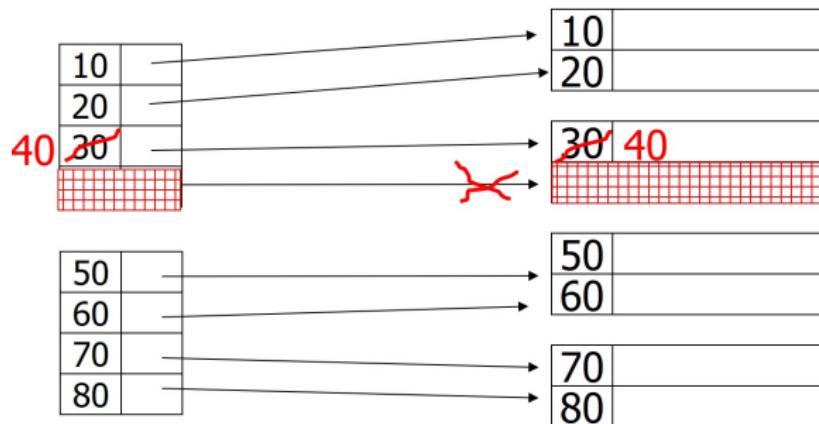
Deletion from dense index

delete record 30

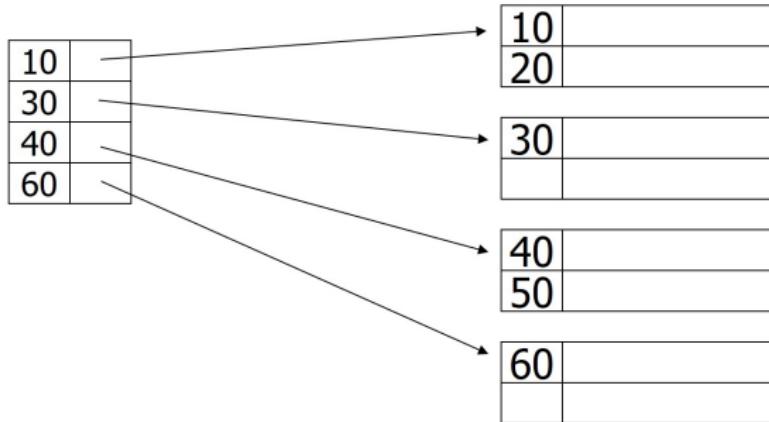


Deletion from dense index

delete record 30

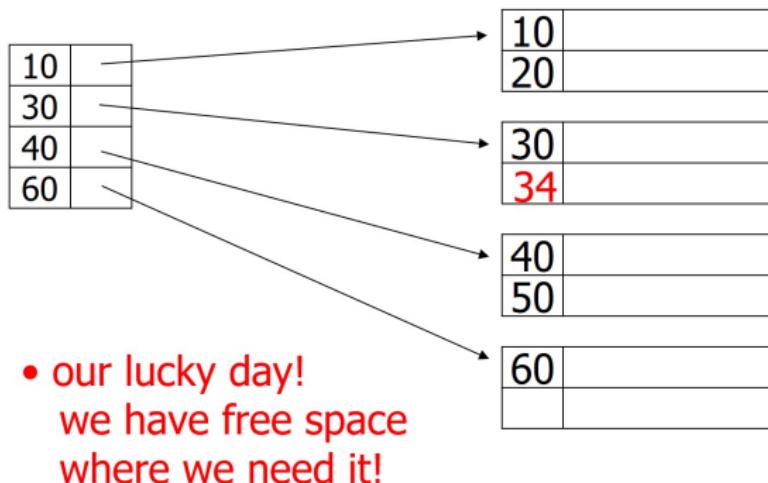


Insertion, sparse index case



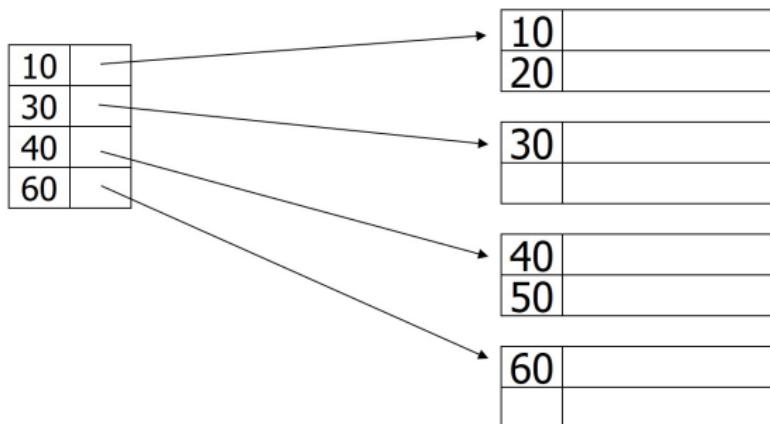
Insertion, sparse index case

insert record 34

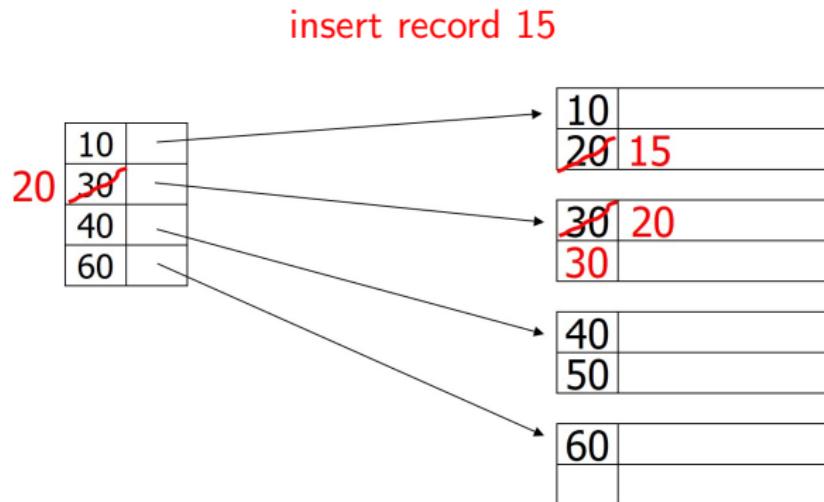


Insertion, sparse index case

insert record 15



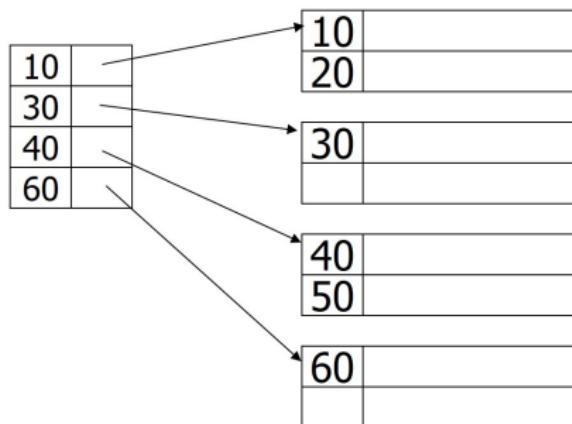
Insertion, sparse index case



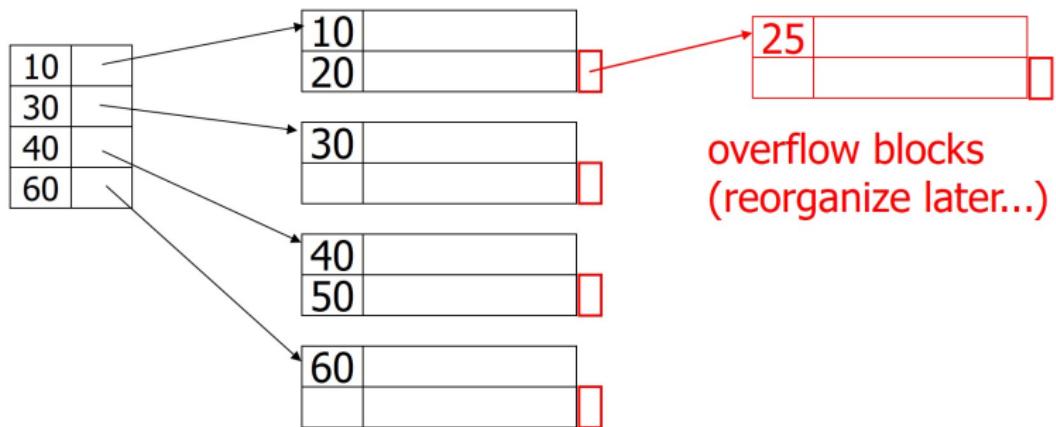
- Illustrated: Immediate reorganization
- Variation
 - insert new block (chained file)
 - update index

Insertion, sparse index case

insert record 25



Insertion, sparse index case



Insertion, dense index case

- Similar
- Often more expensive ...

Topics

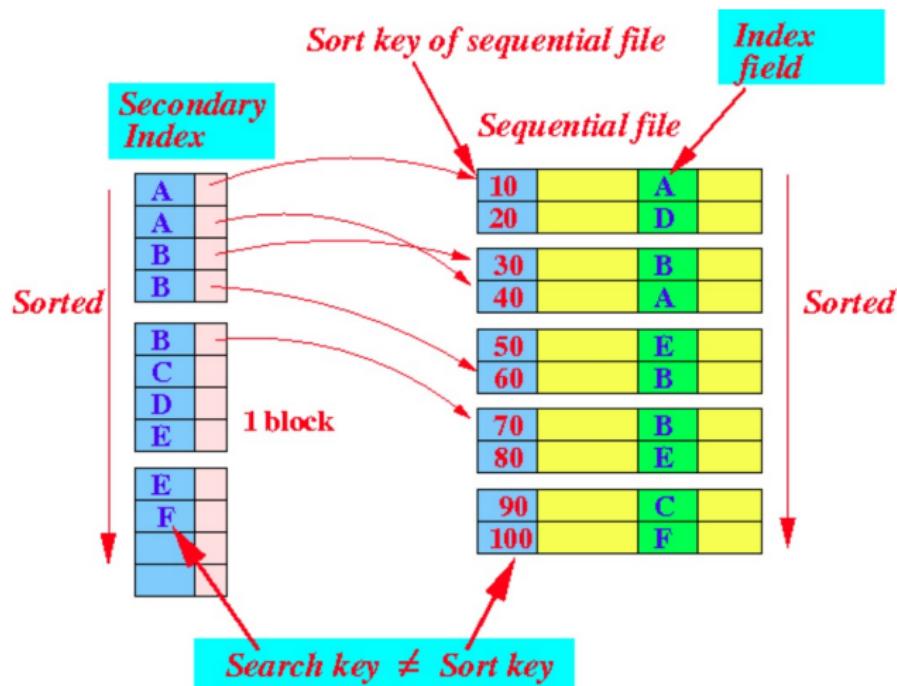
- Conventional indexes
 - Basic Ideas: sparse, dense, multi-level ...
 - Duplicate Keys
 - Deletion/Insertion
 - Secondary indexes
- B-Trees
- Hashing schemes

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?

Sequence
field

30	
50	

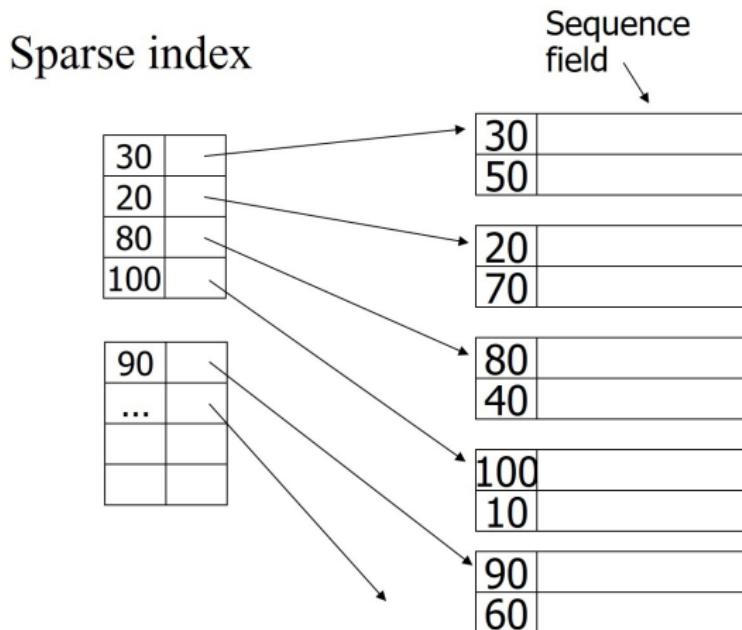
20	
70	

80	
40	

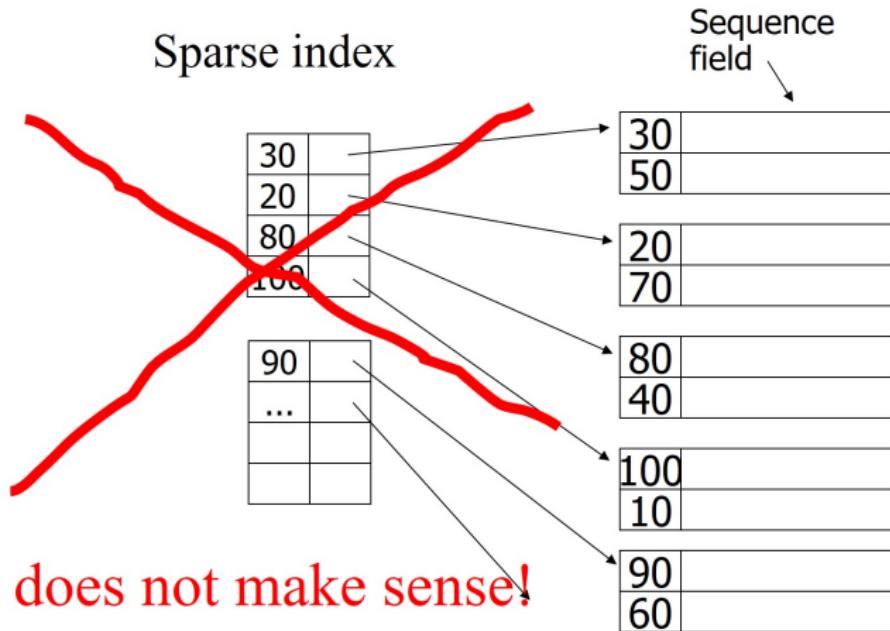
100	
10	

90	
60	

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

Sequence
field

30	
50	

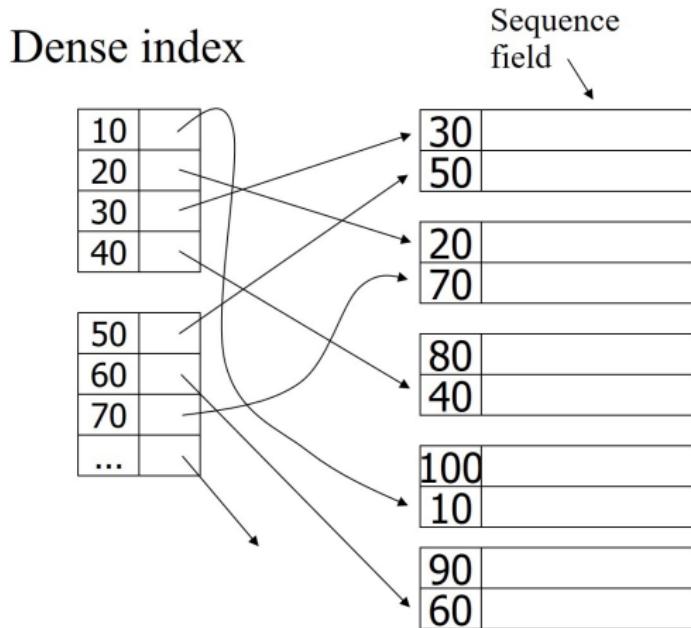
20	
70	

80	
40	

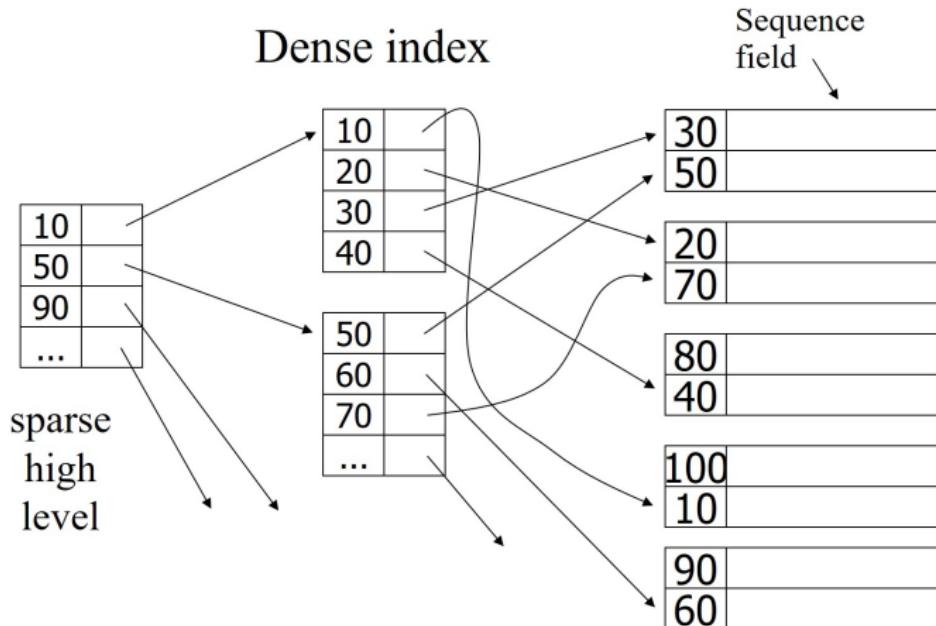
100	
10	

90	
60	

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	

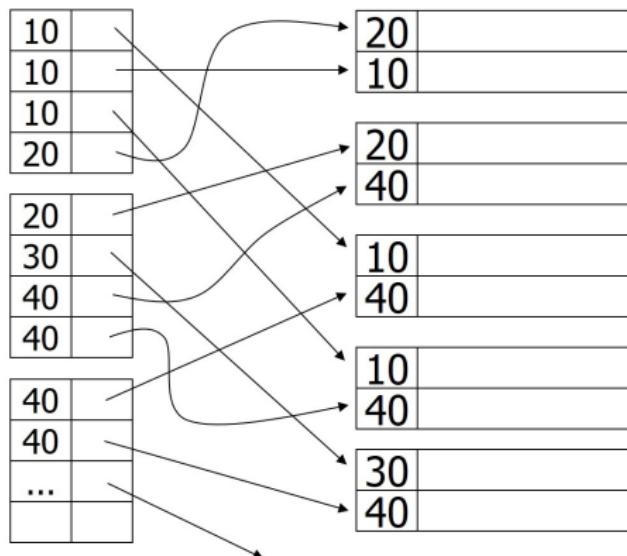
10	
40	

10	
40	

30	
40	

Secondary Index and Duplicate Keys

- one option

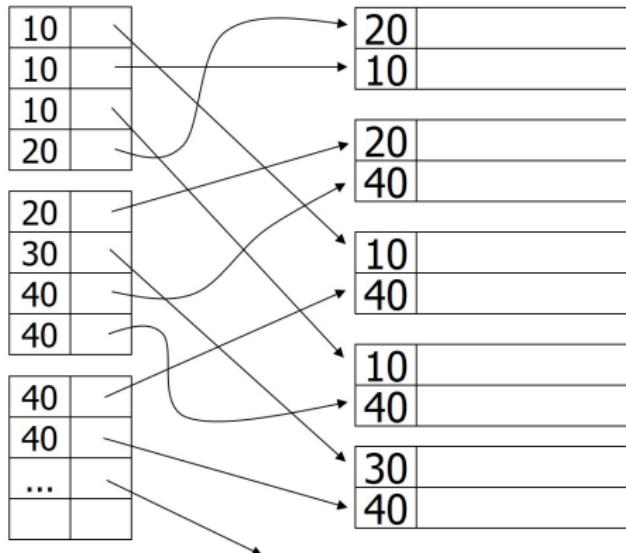


Secondary Index and Duplicate Keys

- one option

Problem:
excess overhead!

- disk space
- search time



Secondary Index and Duplicate Keys

- another option

20	
10	

20	
40	

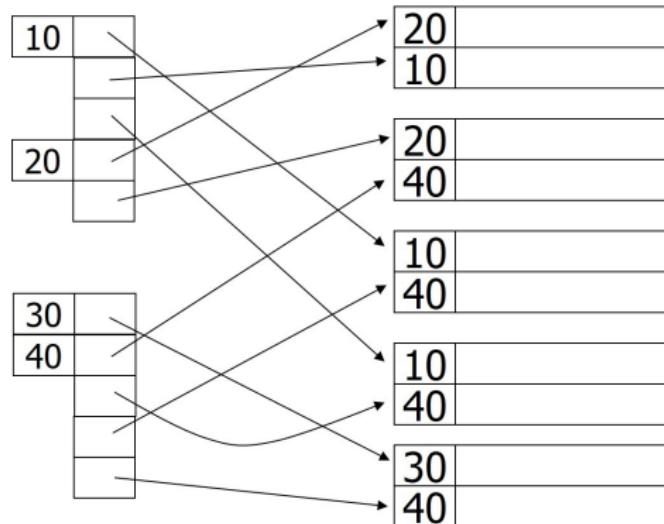
10	
40	

10	
40	

30	
40	

Secondary Index and Duplicate Keys

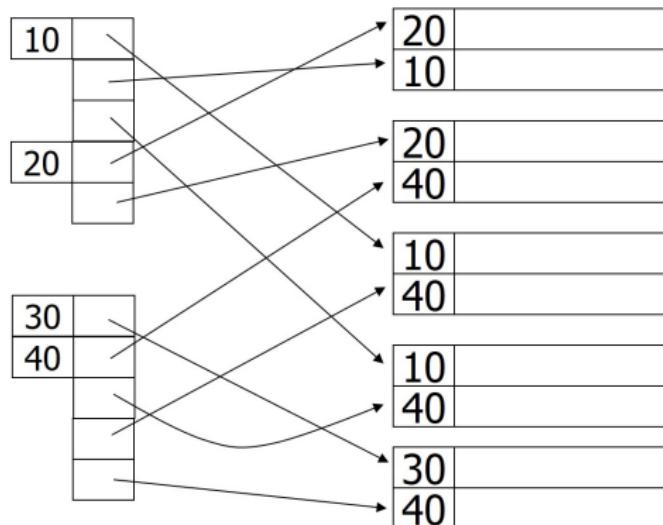
- another option



Secondary Index and Duplicate Keys

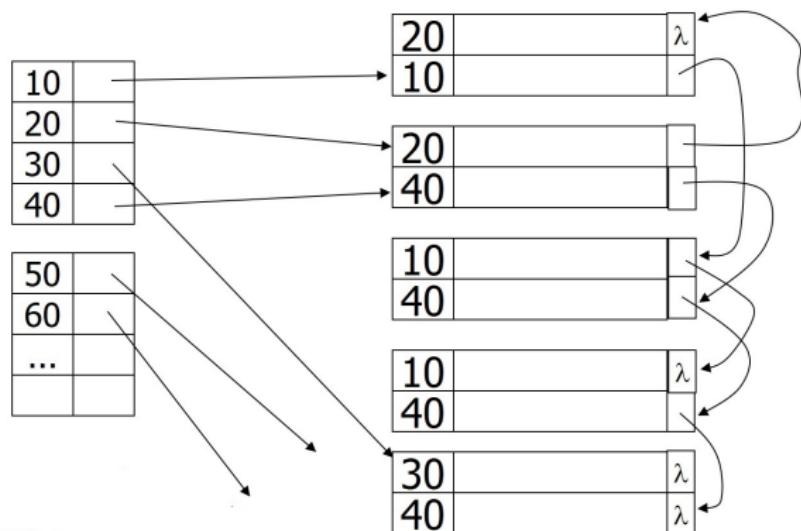
- another option

Problem:
variable size
records in
index!



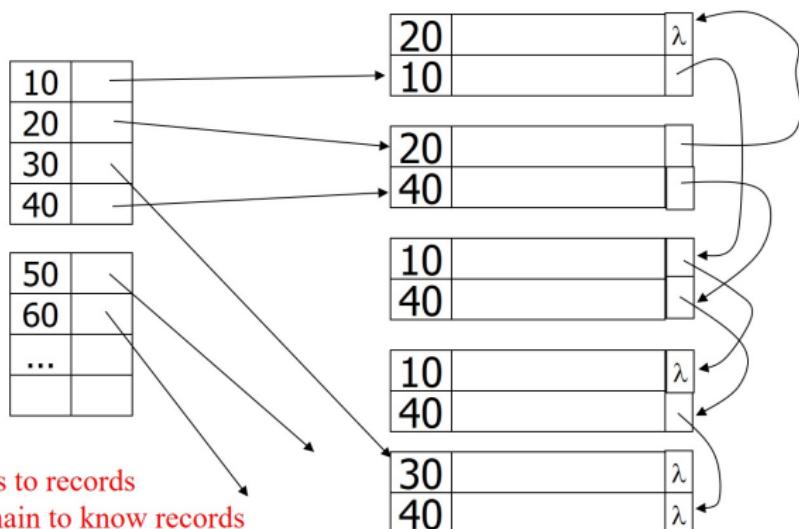
Secondary Index and Duplicate Keys

- another idea: Chain records with same key option



Secondary Index and Duplicate Keys

- another idea: Chain records with same key option



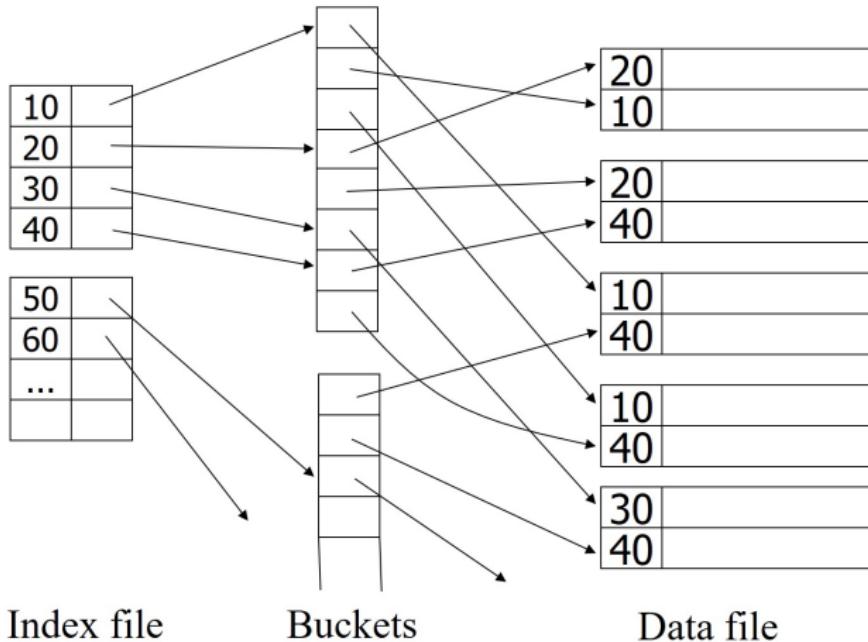
Problems:

- 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

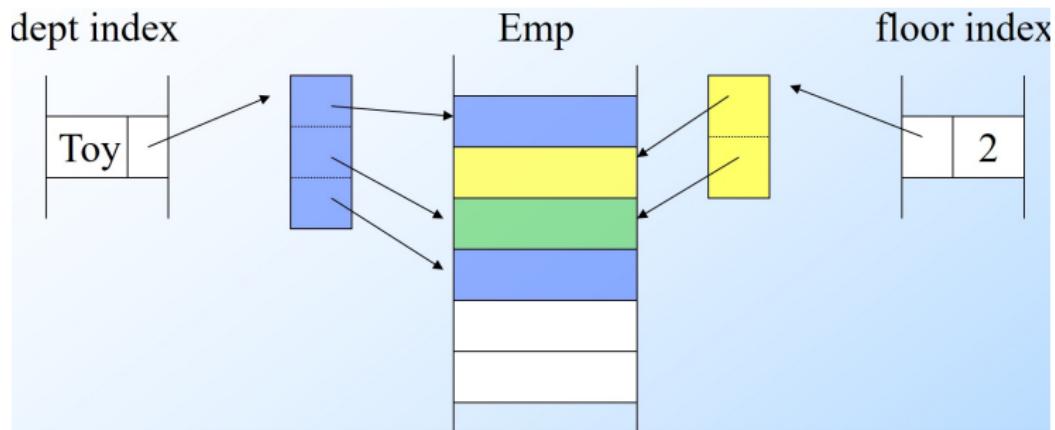
- Saves space as long as search-keys are larger than pointers and average key appears at least twice
- We can use the pointers in the buckets to help answer queries without ever looking at most of the records in the data file.
 - When there are several conditions to a query, and each condition has a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

This idea used in text information retrieval

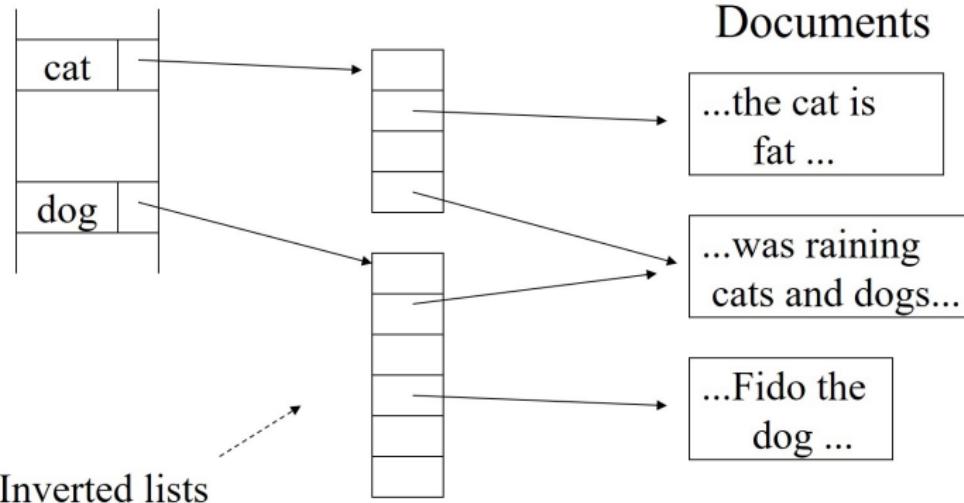
Documents

...the cat is
fat ...

...was raining
cats and dogs...

...Fido the
dog ...

This idea used in text information retrieval

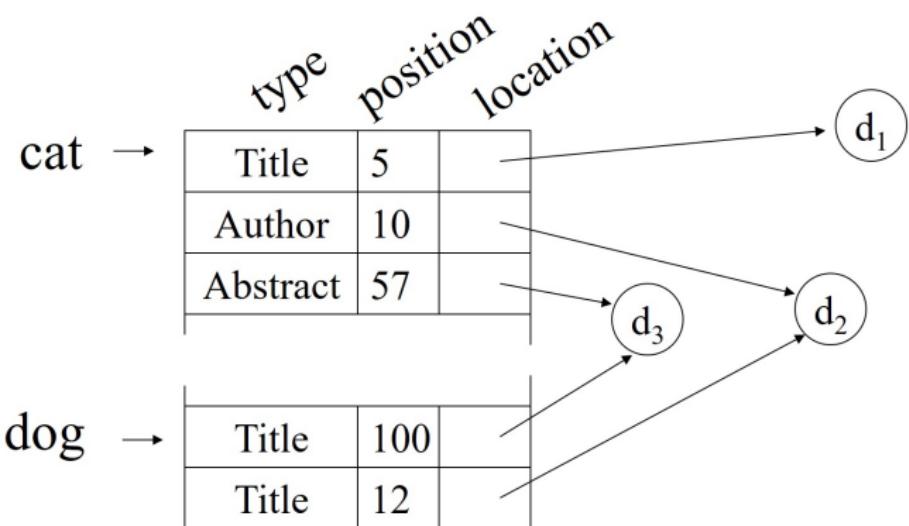


inverted index consists of a set of word-pointer pairs; the words are in effect the search key for the index.

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

Common technique: more info in inverted list



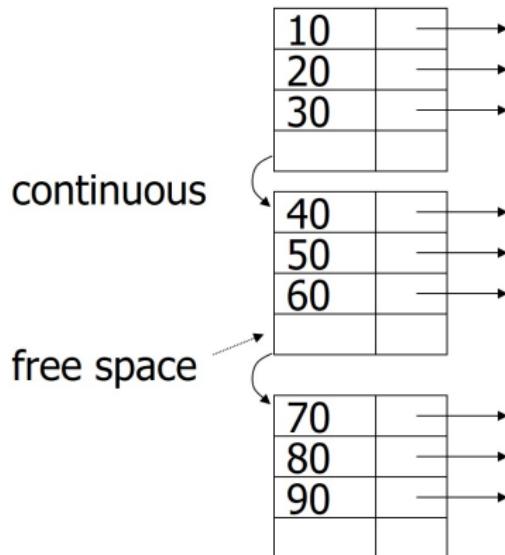
Summary so far

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

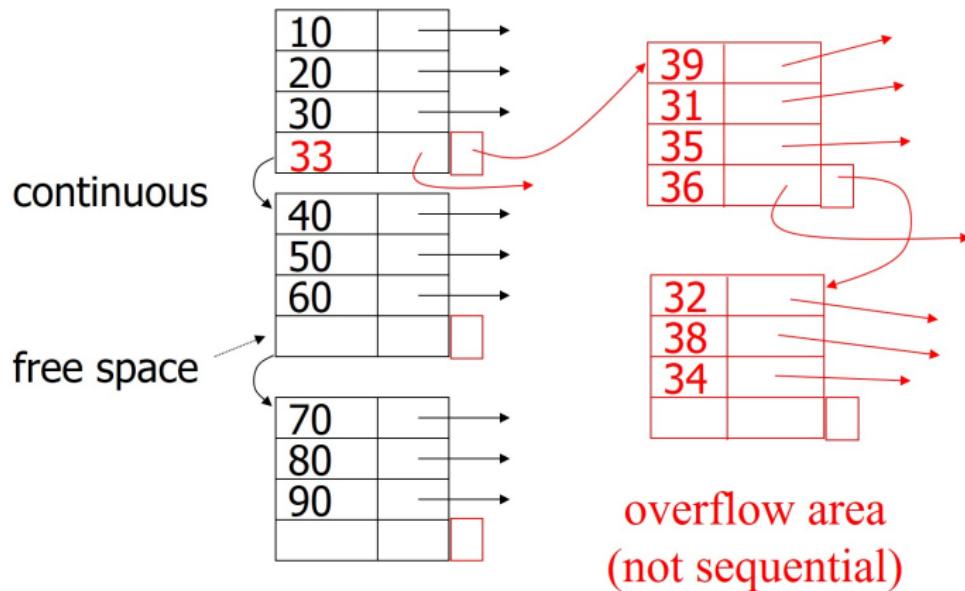
Conventional indexes

- 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

Example: Index (sequential)



Example: Index (sequential)



Next

- B-tree