Today:

- Modifications to indexed-sequential files.
- Secondary indexes, inverted indexes.
- B-trees: a clean way to manage multilevel indexes.

Soon:

- Hashing, another powerful indexing technique.
- Multidimensional index structures.

DB Modifications

When we insert or delete on the data file, here are the primitive actions we might take:

1. Create or destroy an empty block in the sequence of blocks belonging to the sequential file.
2. Create or destroy an overflow block.
3. Insert a record into a block that has room.
4. Delete a record.
5. Slide a record to an adjacent block.

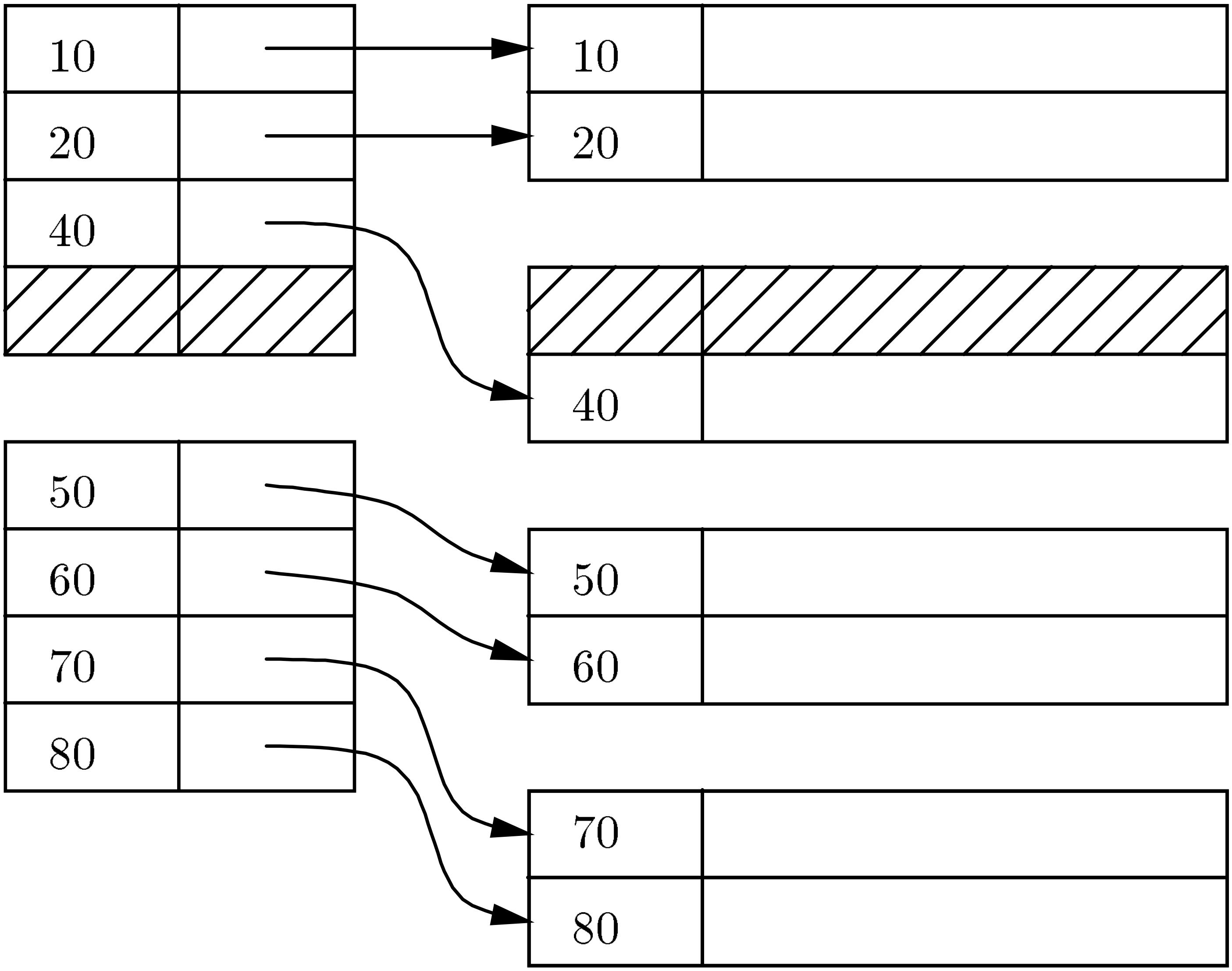
Effect of Primitive Actions on Index File

Action	Dense	Sparse
Create/destroy empty ovflow block	none	none
Create empty seq. block	none	insert
Destroy empty seq. block	none	delete
Insert record	insert	update(?)
Delete record	delete	update(?)
Slide record	update	update(?)

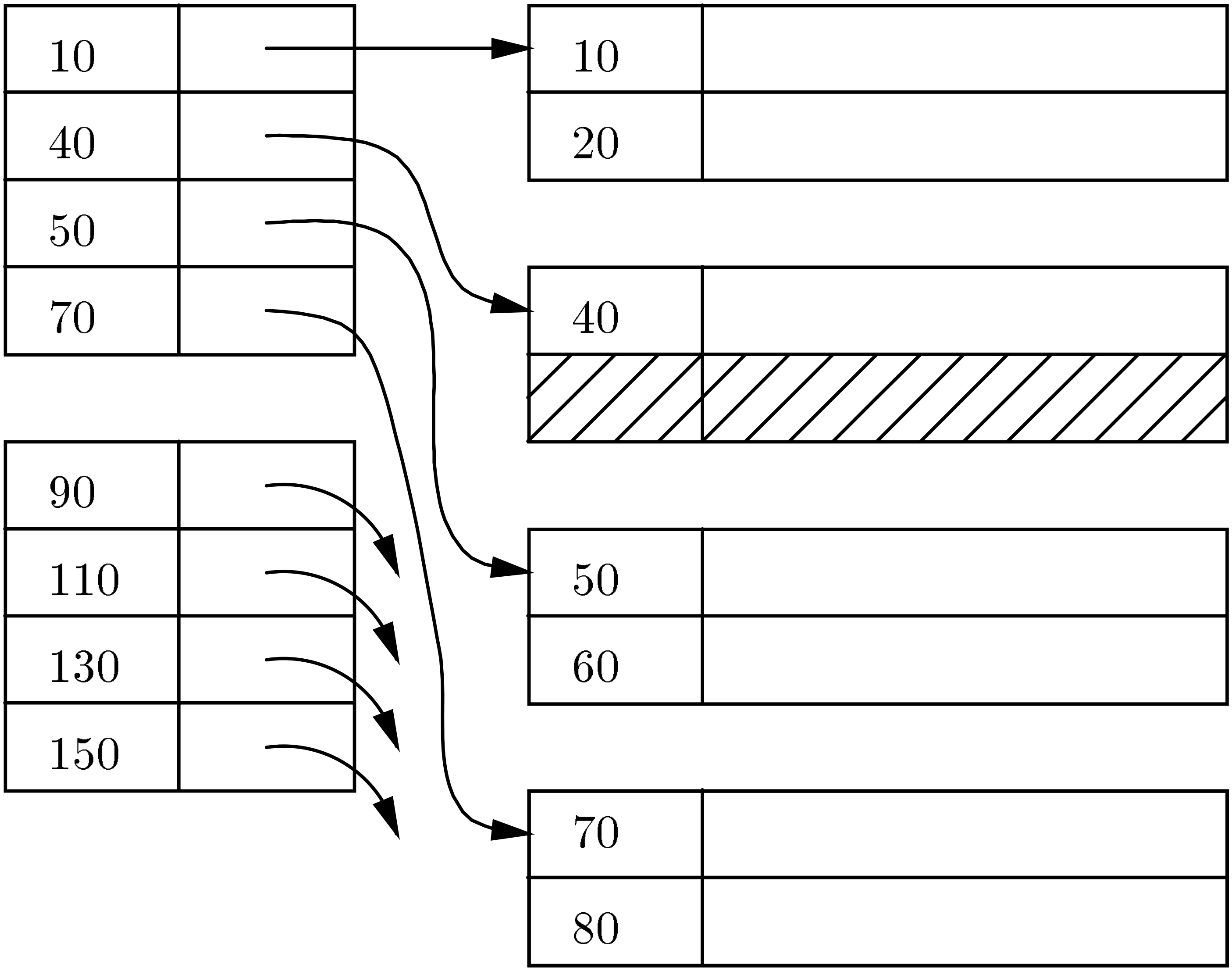
Options

- Compact data and/or index blocks (all empty space at end).
    - Only need to record beginning of available space, rather than empty/full for each record-slot.
    - Compaction a problem if pointers from outside index.
- Add sequential blocks or overflow blocks when space needed.
    - Overflow blocks require no change to index, but make sparse indexes "sparser."
- Redistribute records locally or always go for a new block.
    - Redistribution is extra effort, but keeps blocks more full.

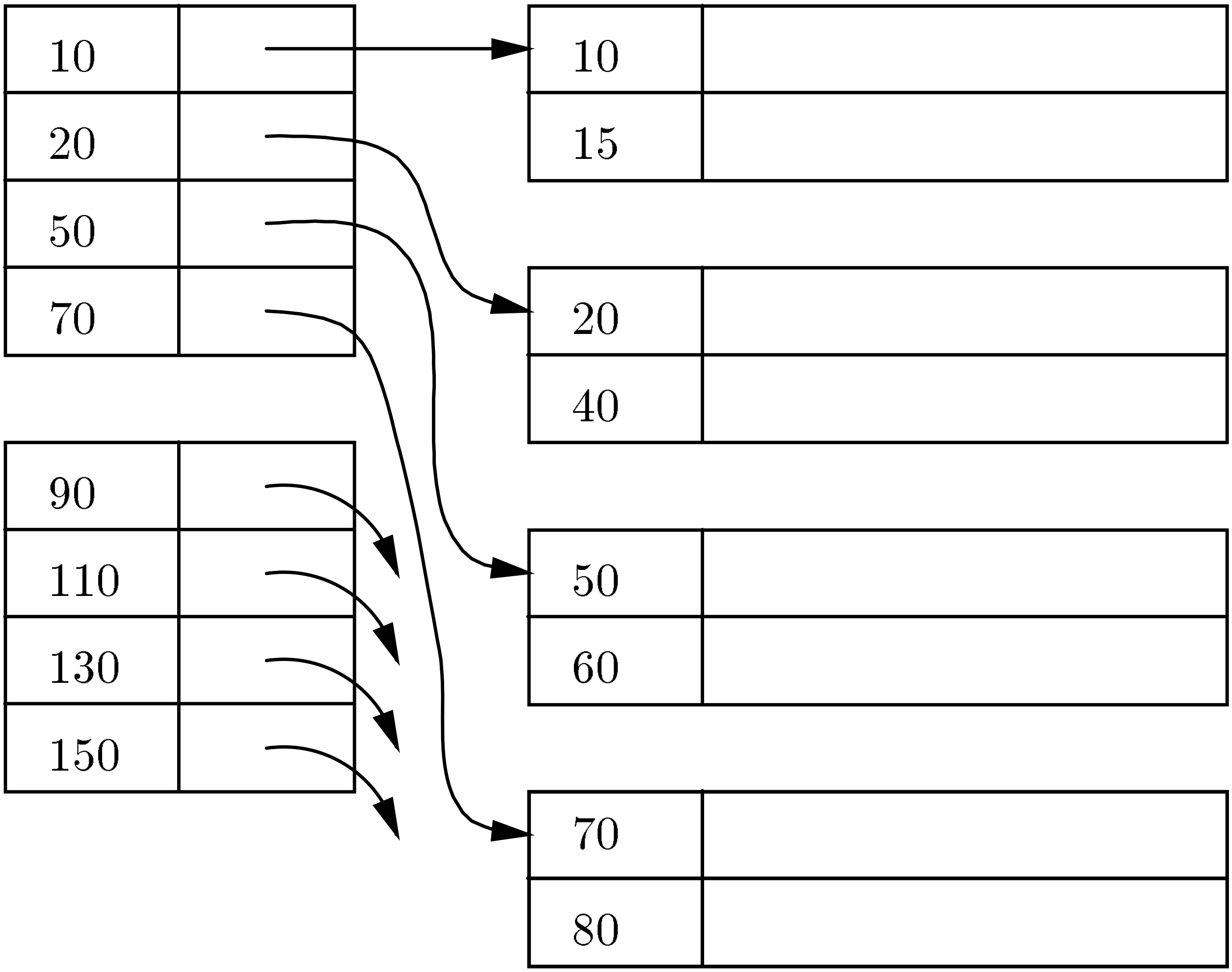
Example: Delete 30 With Dense Index



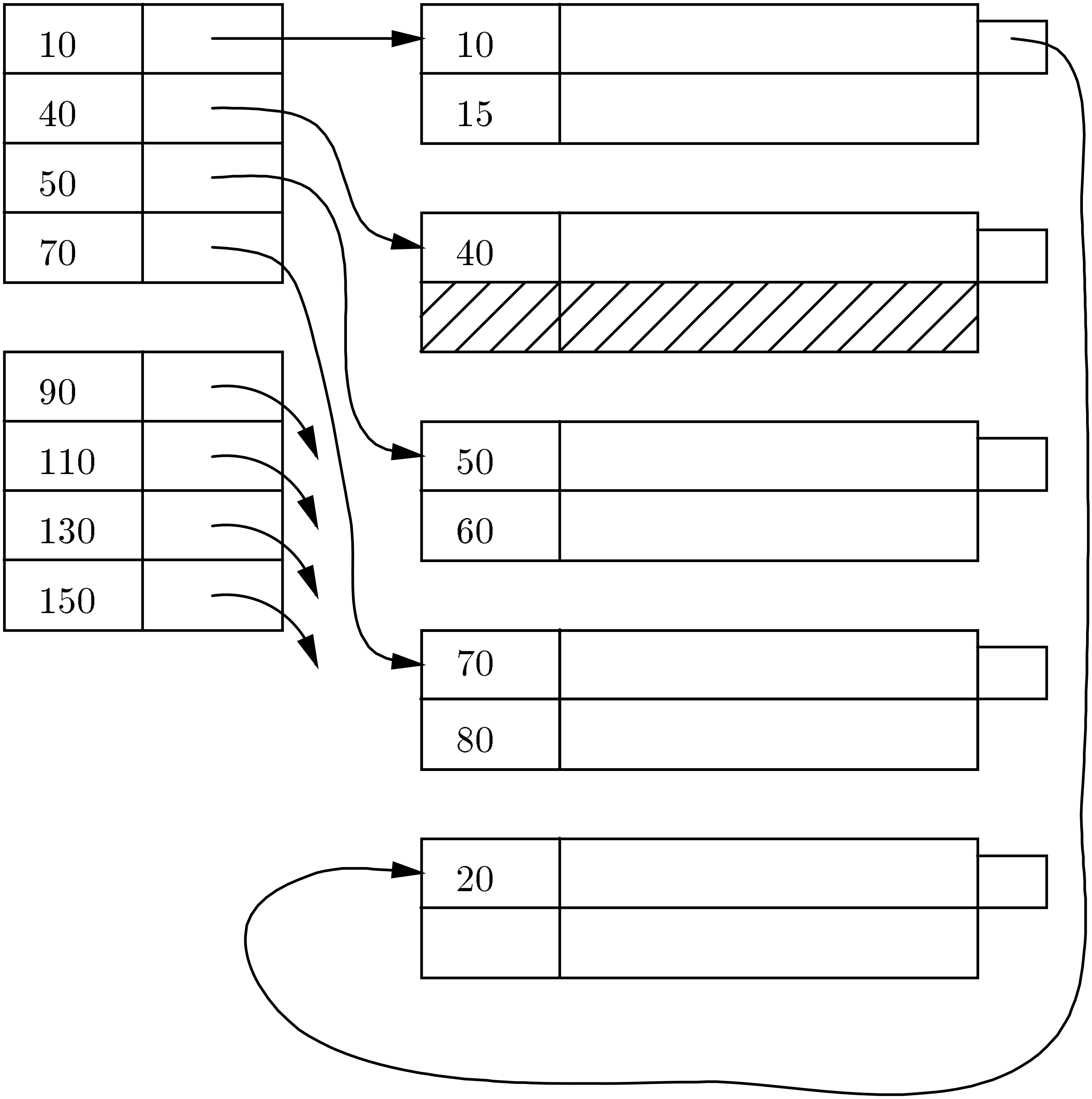
Example: Delete 30 With Sparse Index



Example: Insert 15 With Sparse Index — Redistribute

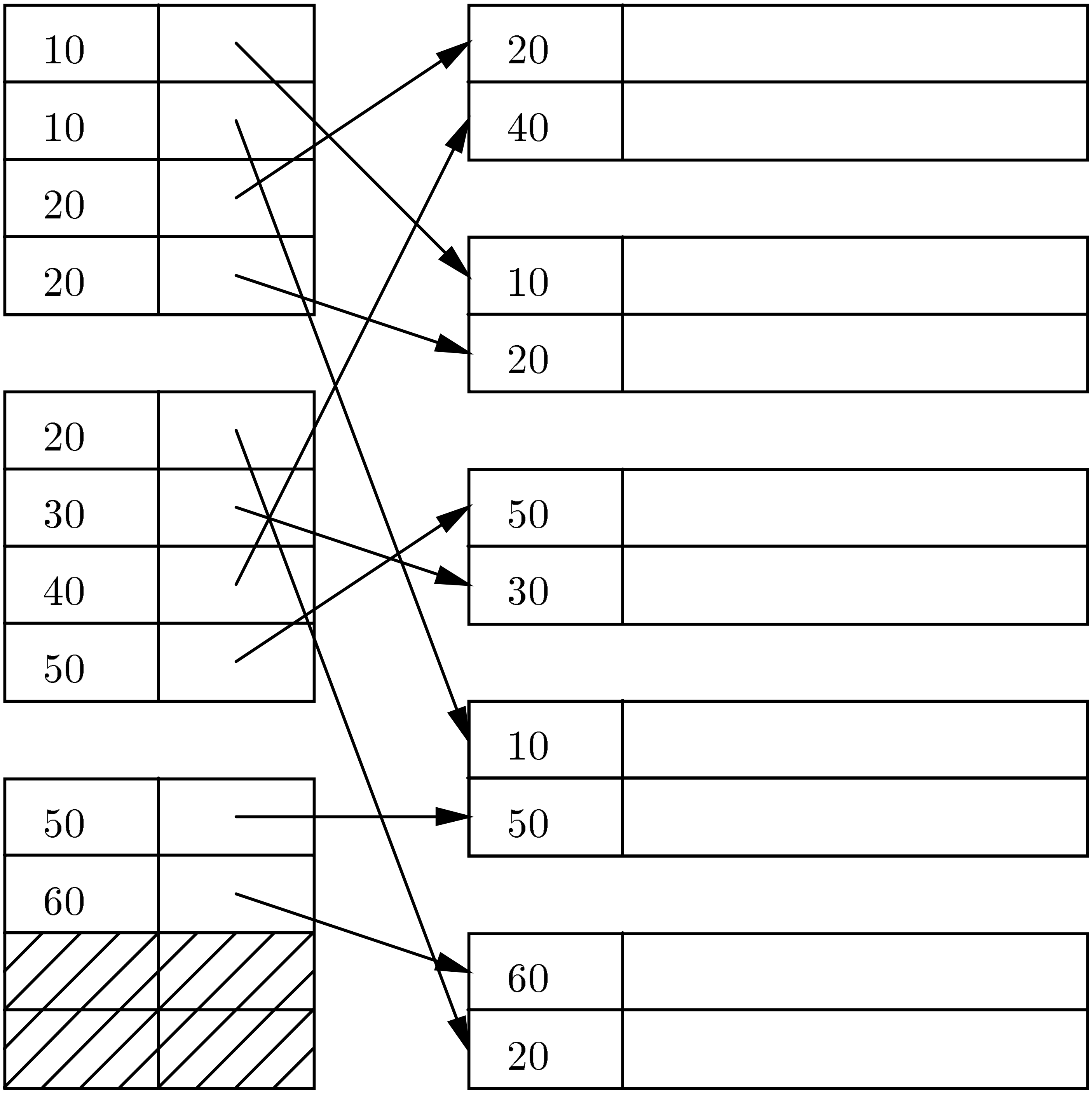


Use Overflow Block Instead



Secondary Indexes

- SKS says *primary index* is an index on a sorted file.
- I prefer to consider any index that "controls" the placement of records to be primary, e.g., hash table.
- *Secondary index* = index that does not control placement, surely not on a file sorted by its search key.
    - ✦ Sparse, secondary index makes no sense.
    - ✦ Usually, search key is not a "key."


10
10
20
20
20
30
40
50
50
60
20
40
10
20
50
30
10
50
60
20

Indirect Buckets

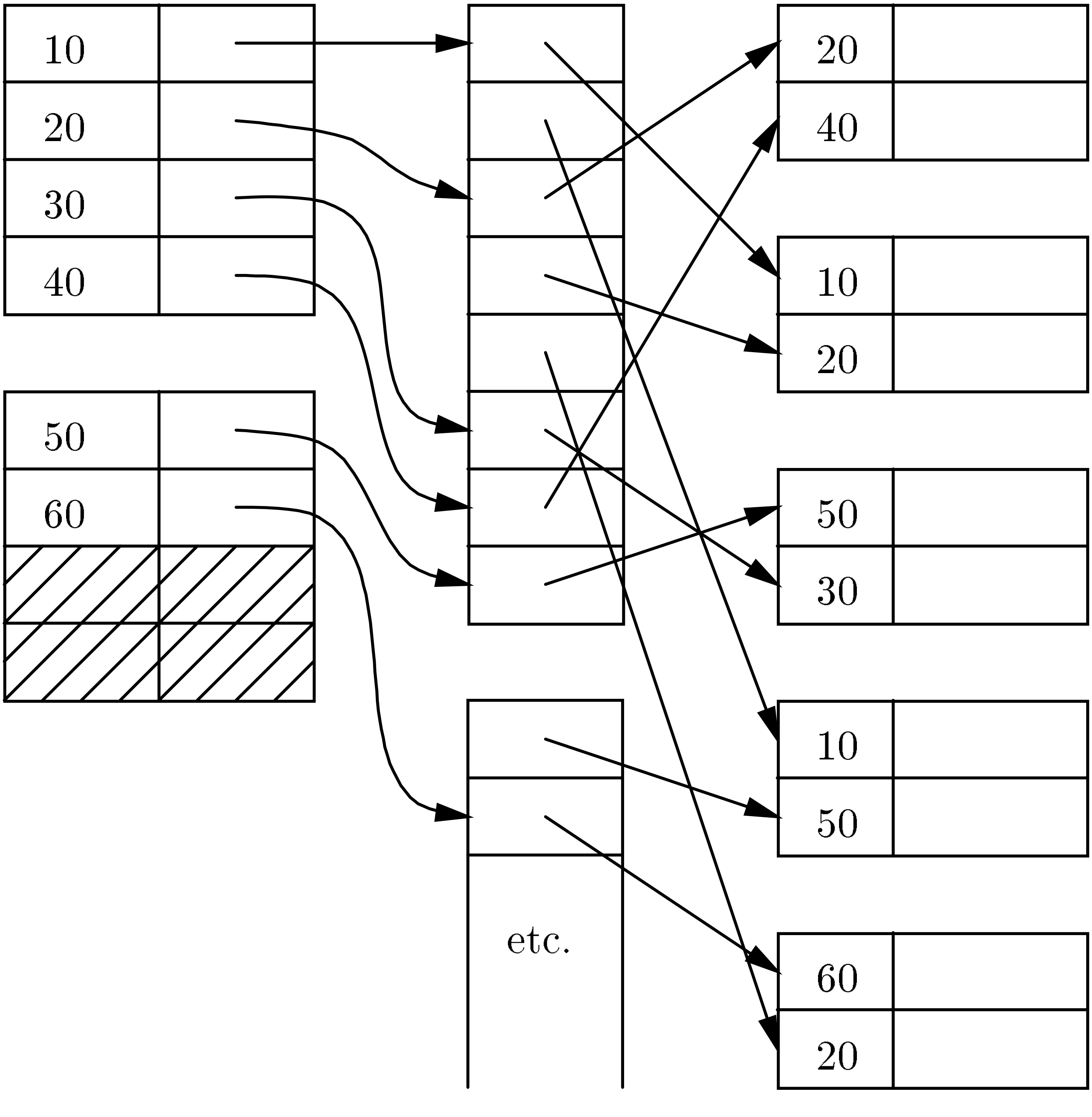
To avoid repeating keys in index, use a level of indirection, called *buckets*.

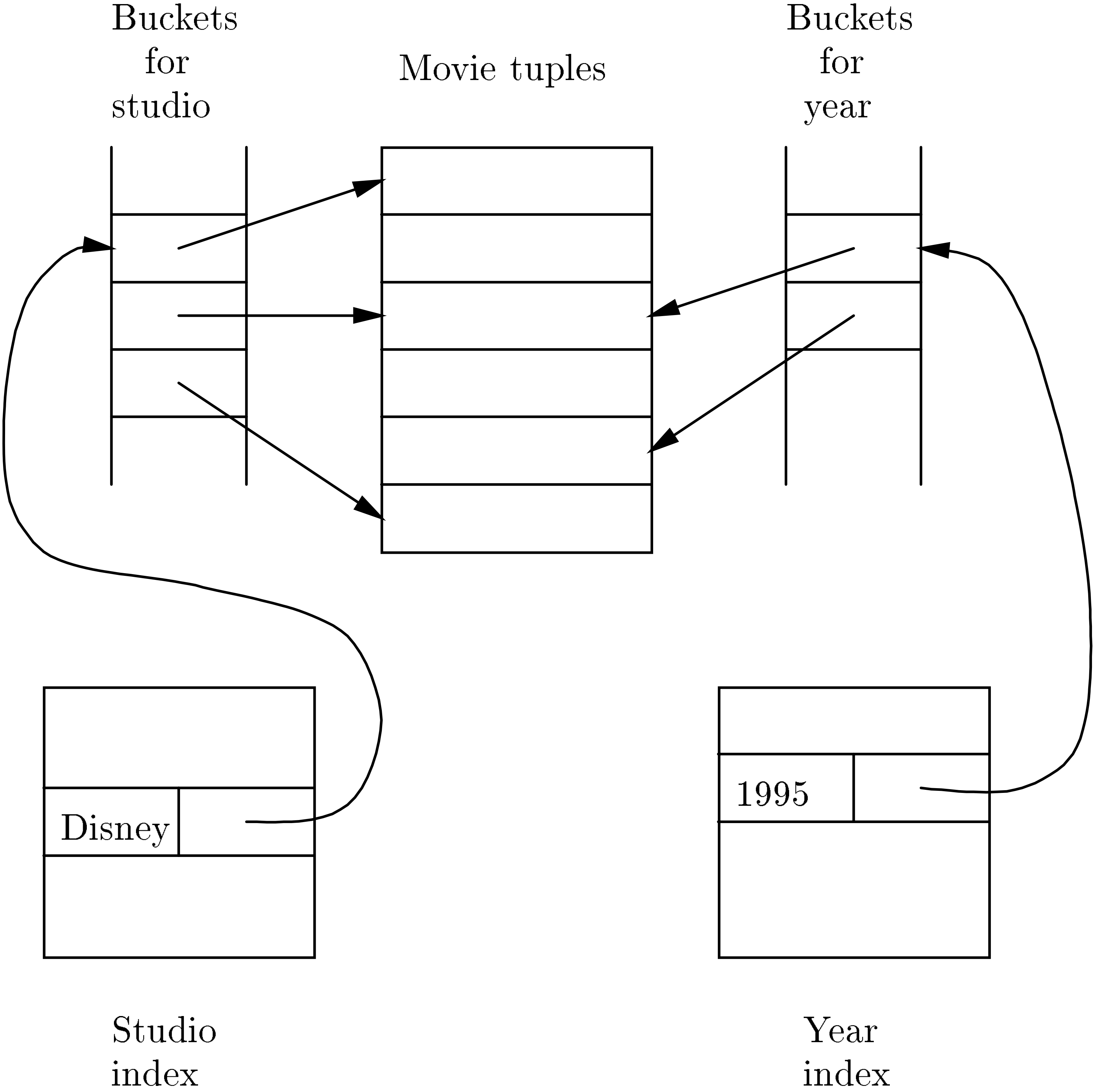
- Additional advantage: allows intersection of sets of records without looking at records themselves.

Example

`Movies(title, year, length, studioName)`; secondary indexes on `studioName` and `year`.

```
SELECT title
FROM Movies
WHERE studioName = 'Disney' AND
    year = 1995;
```


10
20
30
40
50
60
etc.
20
40
10
20
50
30
10
50
60
20

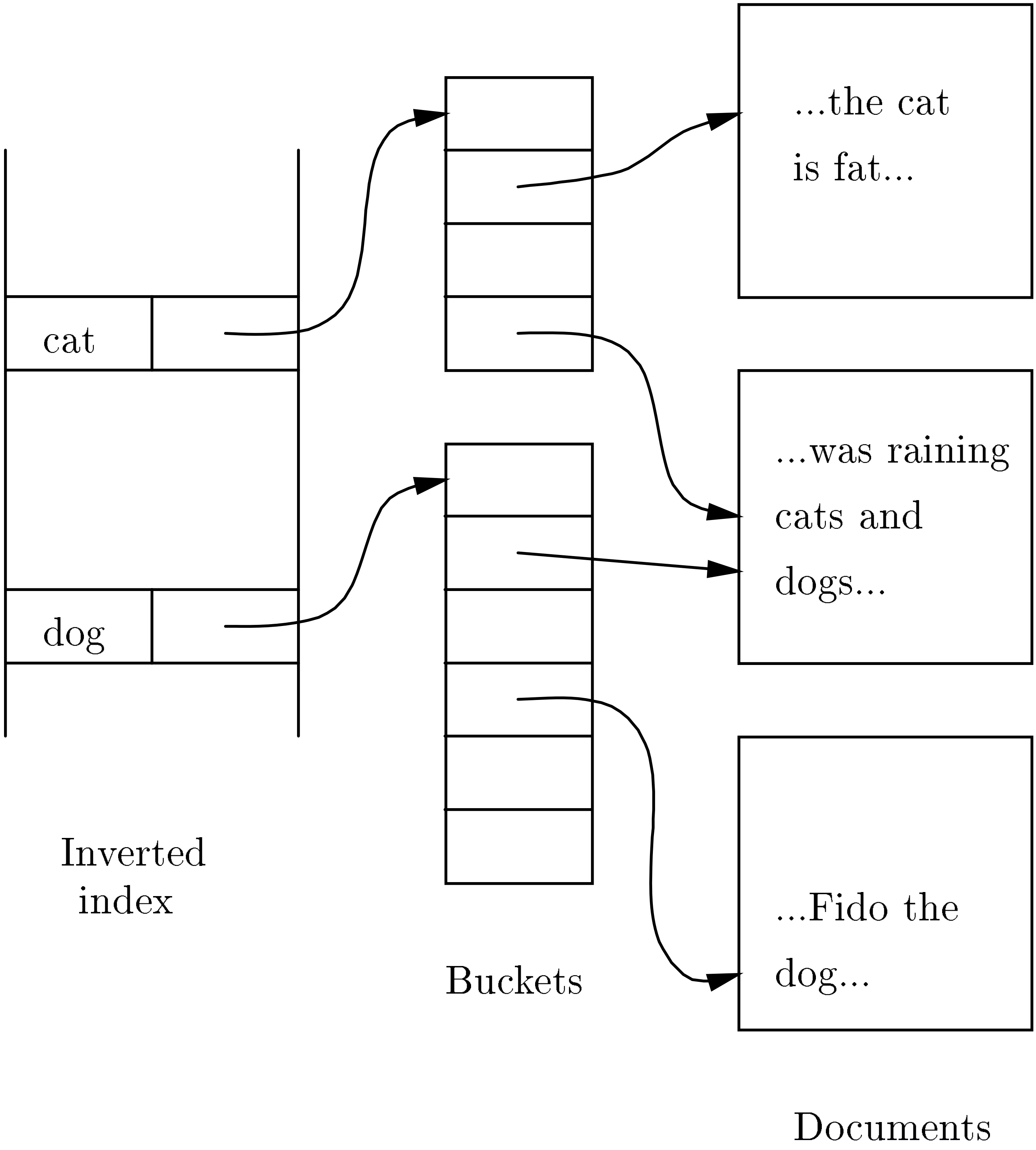

Buckets
for
studio
Movie tuples
Buckets
for
year
Disney
1995
Studio
index
Year
index

Inverted Indexes

Similar (to secondary indexes) idea from information-retrieval community, but:

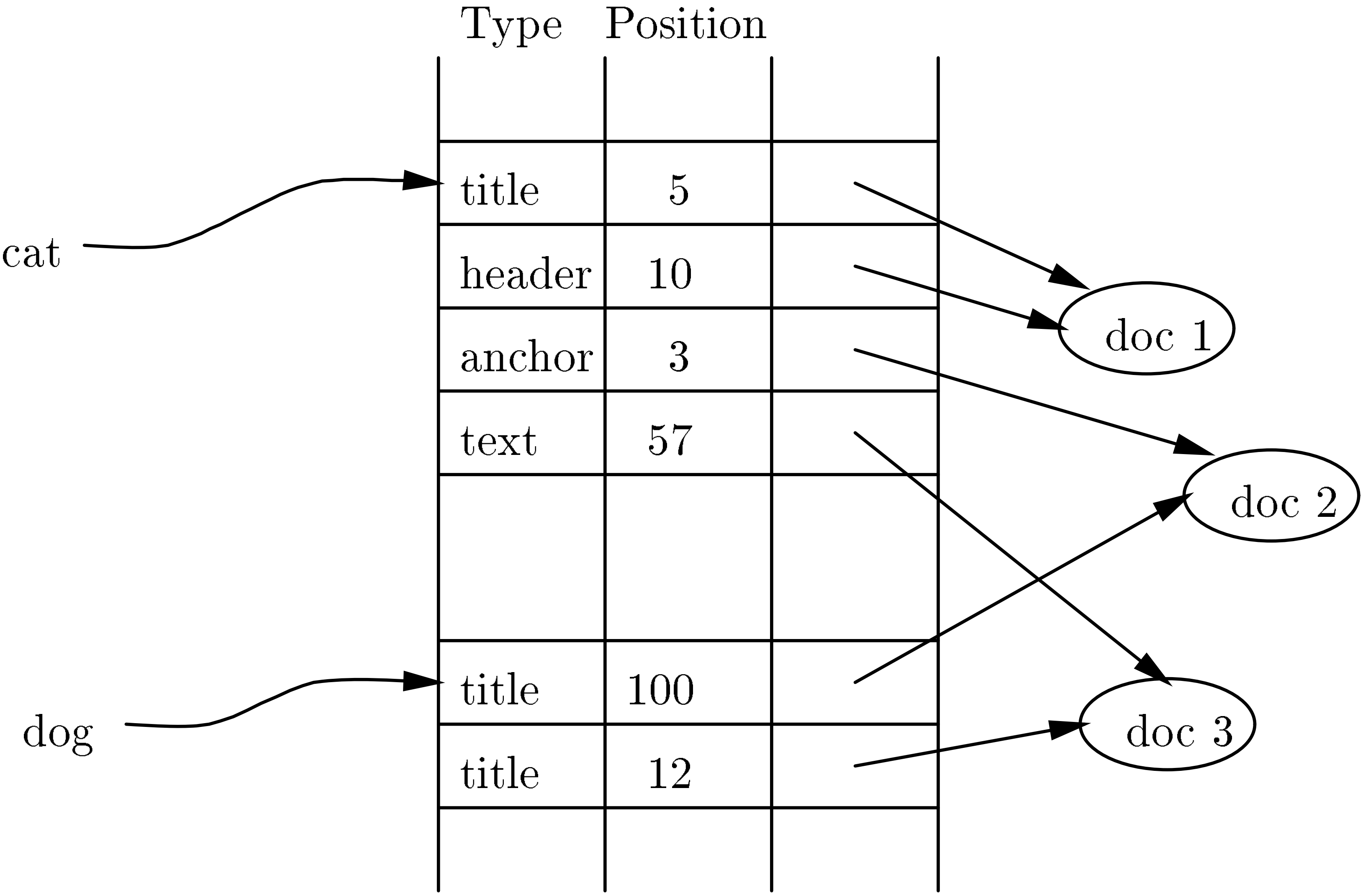
- Record → document.
- Search-key value of record → presence of a word in a document.

Usually used with "buckets."


cat
dog
Inverted index
Buckets
...the cat is fat...
...was raining cats and dogs...
...Fido the dog...
Documents

Additional Information in Buckets

Can extend bucket to include role, position of word, e.g.



Type	Position	
title	5	
header	10	
anchor	3	
text	57	
title	100	
title	12	

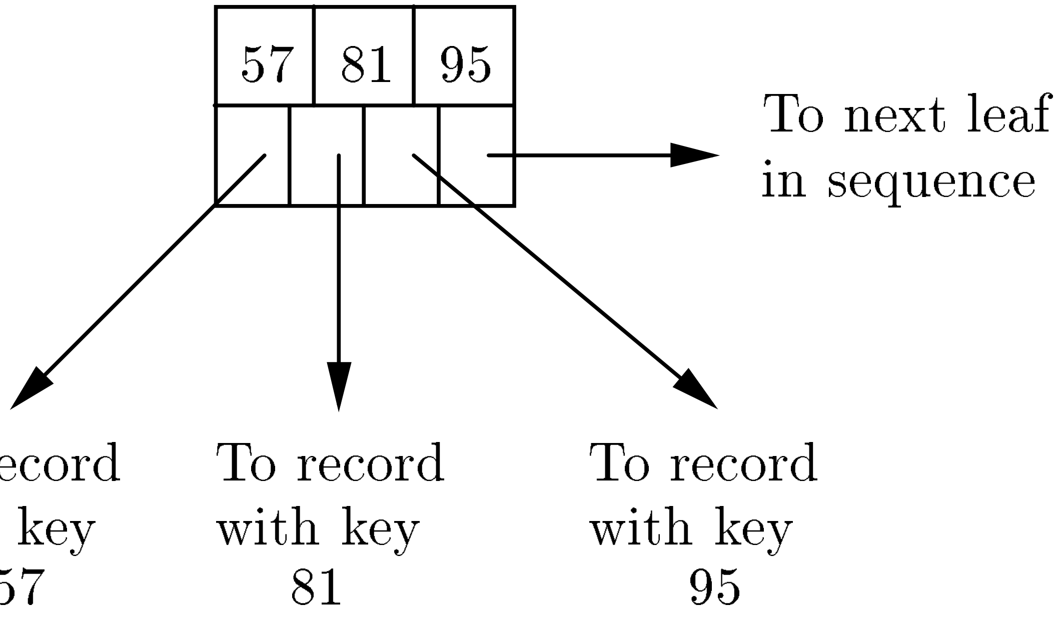
B-Trees

Generalizes multilevel index.

- Number of levels varies with size of data file, but is often 3.
- *B+ tree* = form we'll discuss.
  - ❖ All nodes have same format: n keys, $n+1$ pointers.
- Useful for primary, secondary indexes, primary keys, nonkeys.

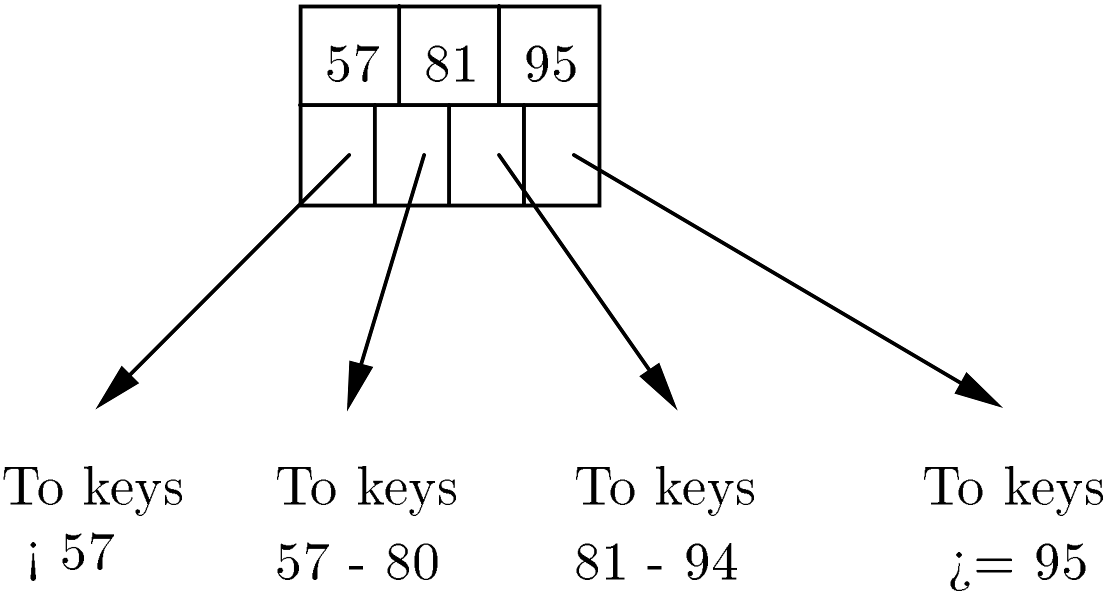
Leaves

- One pointer to next leaf.
- n key-pointer pairs for records of data file.
- At least half of these (round up) occupied.



Interior Nodes

- n keys form the divisions among n+1 subtrees.
  - Key i is least key reachable from $(i + 1)$st child.
- At least $n/2$ (round down) keys used, and one more pointer than key is used.
  - Exception: root may have only 2 children, one key.



To keys ¡ 57

To keys 57 - 80

To keys 81 - 94

To keys ¿= 95

If There are Duplicate Keys

Key i is least *new* key reachable from (i+1)st child.

- Exception: the sole key if there is only one key in that entire subtree.

Lookup in B+ Tree

- Start at root.
- Until you reach a leaf, follow the pointer that could lead to the key you want.
- Search that leaf (and leaves to the right if duplicates are possible).

B+ Tree Insertion

- Search for the key being inserted.
- If there is room for another key-pointer at that leaf, insert there.
- If no room, split leaf.
    - Split of leaf looks like insert of child at level above.
    - Thus, recursive splitting all the way up the tree is possible.
    - Be careful to adjust keys as tree changes.

B+ Tree Deletion

- Search for key being deleted.
- If found, delete.
- If the lower limit on occupancy is violated:
    - First look for an adjacent leaf that is above lower limit; “steal” a key-pointer pair from that leaf.
    - If none, then there must be two adjacent leaves, one at minimum, one below minimum. Just enough to merge nodes.
    - Merger looks like delete above, so recursive deletion possible.
    - Again, make sure keys are adjusted above.
- Sometimes, it is OK to allow a B-tree leaf to become subminimum — no mergers.