Today:

- More about fixed-length records: clustering.
- Database pointers: very large address spaces, pointer swizzling.
- Variable-length records: indirection, split records.
- Indexed-sequential files, simple indexes.

Soon:

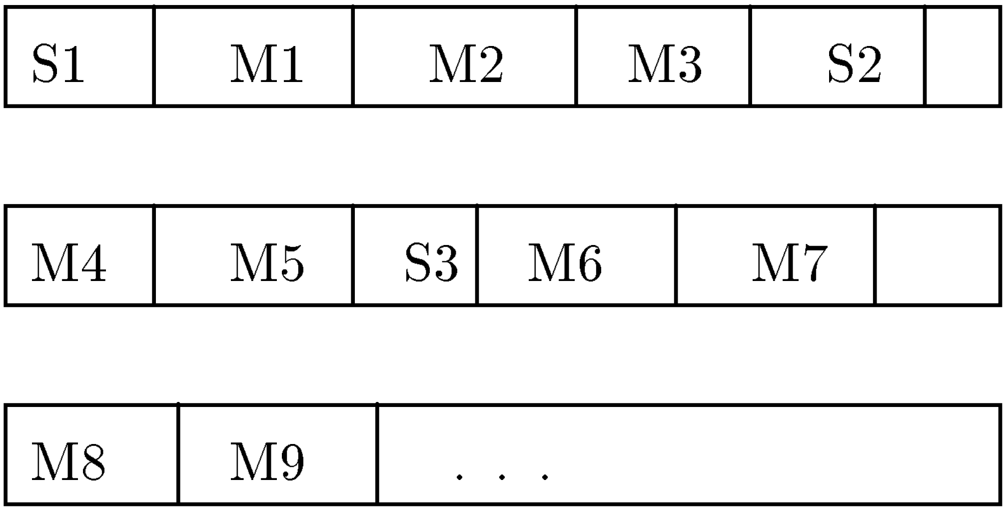
- Better index structures: B-trees, hashing.
- Multidimensional indexing.

Clustering

Suppose there is a many-one relation from relation R to S. It may make sense to keep tuples of R with their "owner" in S.

Example

```
Studios(name, addr)
Movies(title, year, studio)
```



Why Cluster?

Supports queries such as:

```
SELECT title
FROM Movies
WHERE studio = 'Disney';
```

- Few disk I/O's needed to get all Disney movies.
- Even supports a query in which only the address of the studio is given and a join is needed.
- Notice the importance of type info in record headers.

Problem

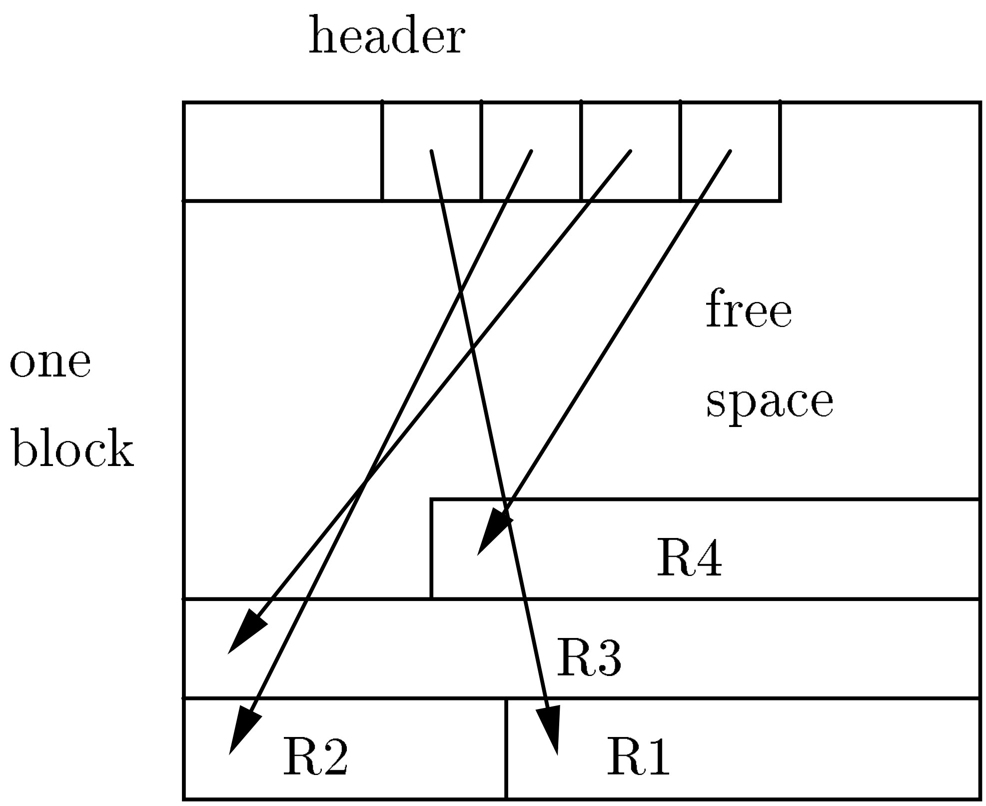
When does clustering lose?

Pointers

- Need pointers to records or blocks:
  - Variable-length records split among blocks, need links.
  - Index data structures link blocks in trees, lists, etc.
  - Relationships, e.g., in ODL.
- 4Gb address space insufficient for referring to record or block addresses throughout a large database. Two approaches:
  1. *Physical*: sequence of bytes composed of device ID, cylinder #, surface #, block # within track, offset within block (for records).
  2. *Logical* (indirect): a *map table* associates abstract ID's, perhaps fixed-length character strings, with physical addresses.

Pointers (Cont.)

- The map table is itself a relation; needs index on ID.
- Physical addresses more efficient.
- Logical addresses allow flexibility: records can move or be deleted without dangling pointers.
  - ✥ Common compromise: physical to block level, table of record offsets within blocks.



Pointer Swizzling

Typical DB structure:

- Data maintained by *server* process, using physical or logical addresses of perhaps 8 bytes.
- Application programs are *clients* with their own (conventional, virtual-memory) address spaces.
- When blocks and records are copied to client's memory, DB addresses must be *swizzled* = translated to virtual-memory addresses.
    - Allows conventional pointer following.
    - Especially important in OODBMS, where pointers-as-data are common.

Swizzling Options

1. *Never swizzle.* Keep a *translation table* of DB pointers → local pointers; consult to follow any DB pointer.

    - Problem: time to follow pointers.

2. *Automatic swizzling.* When a block is copied to memory, replace all its DB pointers by local pointers.

    - Problem: requires knowing where every pointer is (use block and record headers for schema info).

    - Problem: large investment if not too many pointer-followings occur.

3. *Swizzle on demand.* When a block is copied to memory, enter its own address and those of member records into translation table, but do not translate pointers within the block. If we follow a pointer, translate it the first time.

    - Problem: requires a bit in pointer fields for DB/local, extra decision at each pointer following.

Returning Blocks to Disk

- Beware *pinned* blocks or records = some pointer to them is now swizzled.
- Pointers in the returned block must be unswizzled.
- Locate swizzled pointers to block (linked list in translation table?) and unswizzle.

Variable-Length Records

- Fields that vary in length.
- *Repeating fields*, e.g., a set of pointers represent a many-many relationship.
- *Variable-format records*: field names are arbitrary.
  - Important for self-describing data, information integration.

Sliding Records

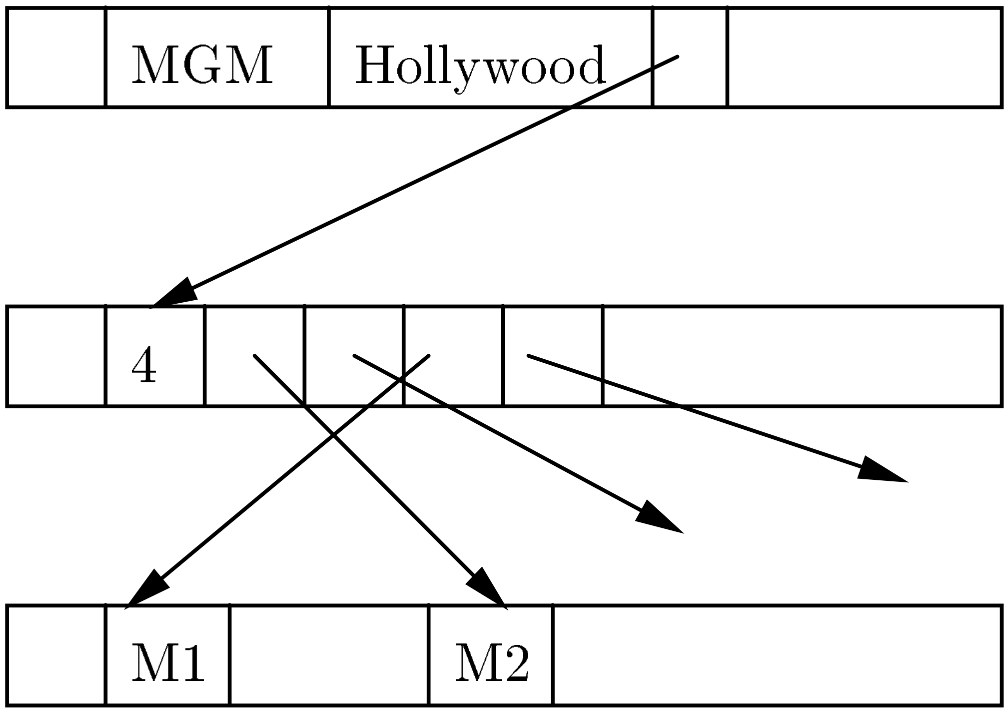
- Use offset table in a block, pointing to current records.
- If a record grows, slide records around the block.
  - Not enough space? Create overflow block; offset table must indicate "record moved."

Split Records Into Fixed/Variable Parts

Fixed part has a pointer to space where current value of variable fields can be found.

Example

Studio records = name and address (fixed), plus a set of pointers to movies by that studio.



BLOB's

Binary, Large Object = field whose value doesn't fit in a block, e.g., video clip.

- Hold in collection of blocks, e.g., several cylinders for fast retrieval.
- Allow client to access only part of value, e.g., get first 10 seconds of a video, and supply each 10-second segment within next 10 seconds.

Self-Describing Records

- Fields are alternating attribute names (pointers to schema?) and values.
- Record header includes length, perhaps schema info.

Example

name	Sue	addr	Maple St.	addr	Beach	

Files

Collection of blocks. How located?

1. Linked list of blocks.
2. Tree-structured access as for UNIX files.
3. Other index structures — to be covered in detail.

Sequential Files

- Records ordered by *search key* (may not be "key" in DB sense).
- Blocks containing records therefore ordered.
- On insert: put record in appropriate block if room.
  - ❖ Good idea: initialize blocks to be less than full; reorganize periodically if file grows.
- If no room in proper block:
  1. Create new block; insert into proper order if possible (what if blocks are consecutive around a track for efficiency?).
  2. If not possible, create *overflow block*, linked from original block.

  In either case, distribute records between the two blocks.

Deletion/Tombstones

- On delete: can we remove record, possibly consolidate blocks, delete overflow blocks?
  - Danger: pointers to record become dangling.
  - Solution: *tombstone* in record header = bit saying deleted/not. Rest of record space can be reused.

Dense Indexes

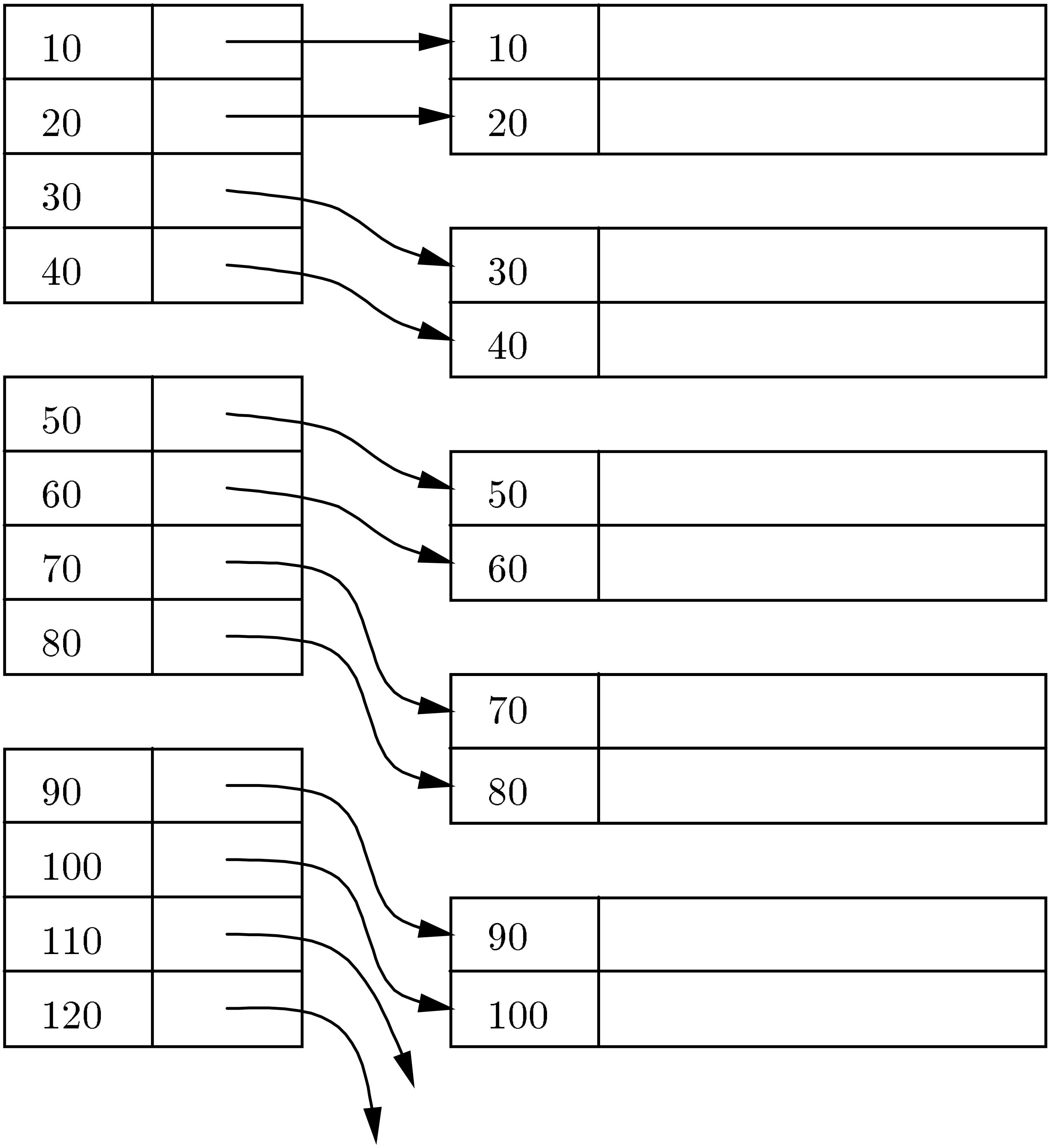
Pointer to every record of file, ordered by search key.

- Can make sense because records may be much bigger than key-pointer pairs.
  - Fit index in memory, even if data file does not? Faster search through index than data file?
  - Test existence of record without going to data file.

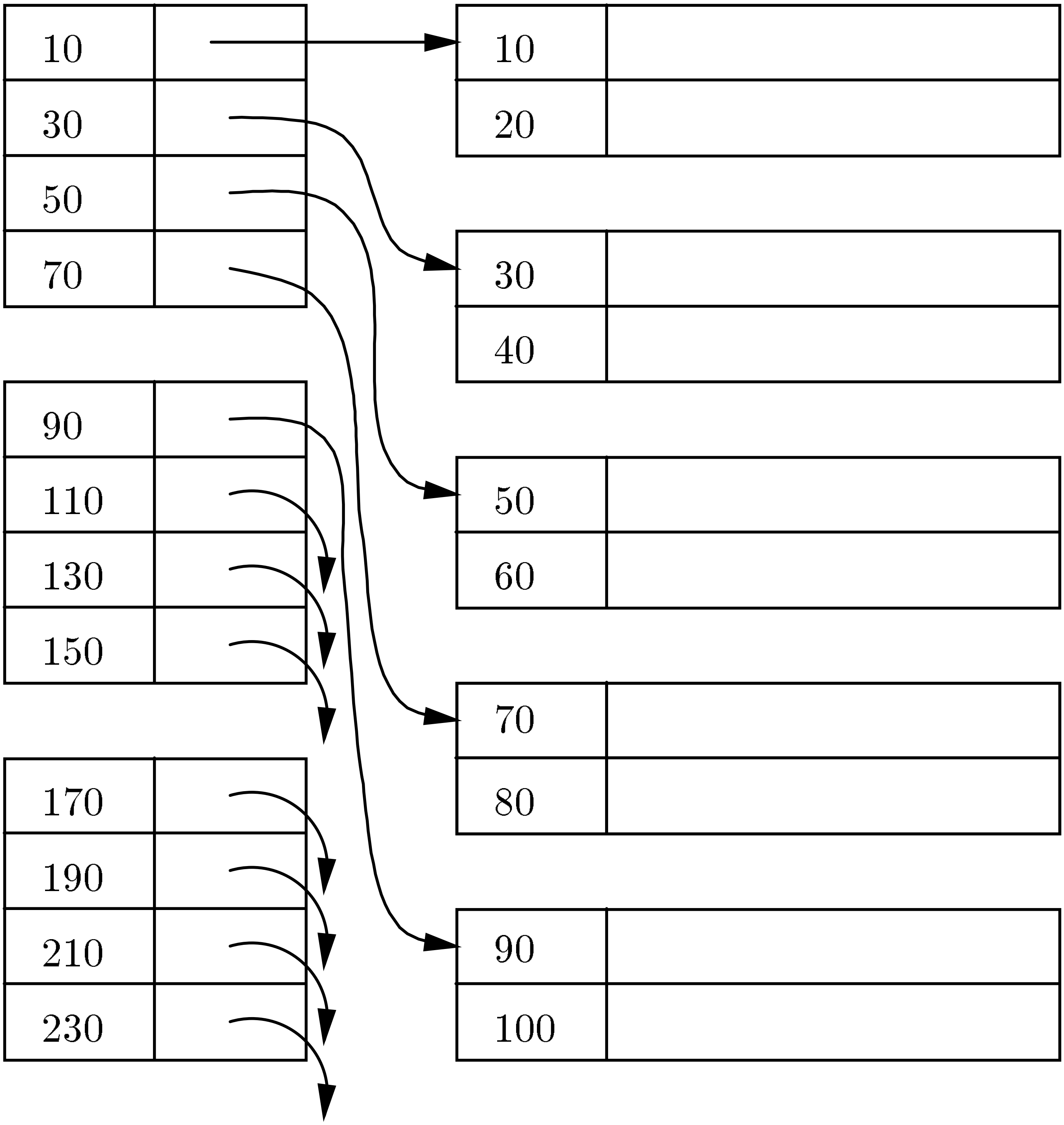
Sparse Indexes

Key-pointer pairs for only a subset of records, typically first in each block.

- Saves index space.



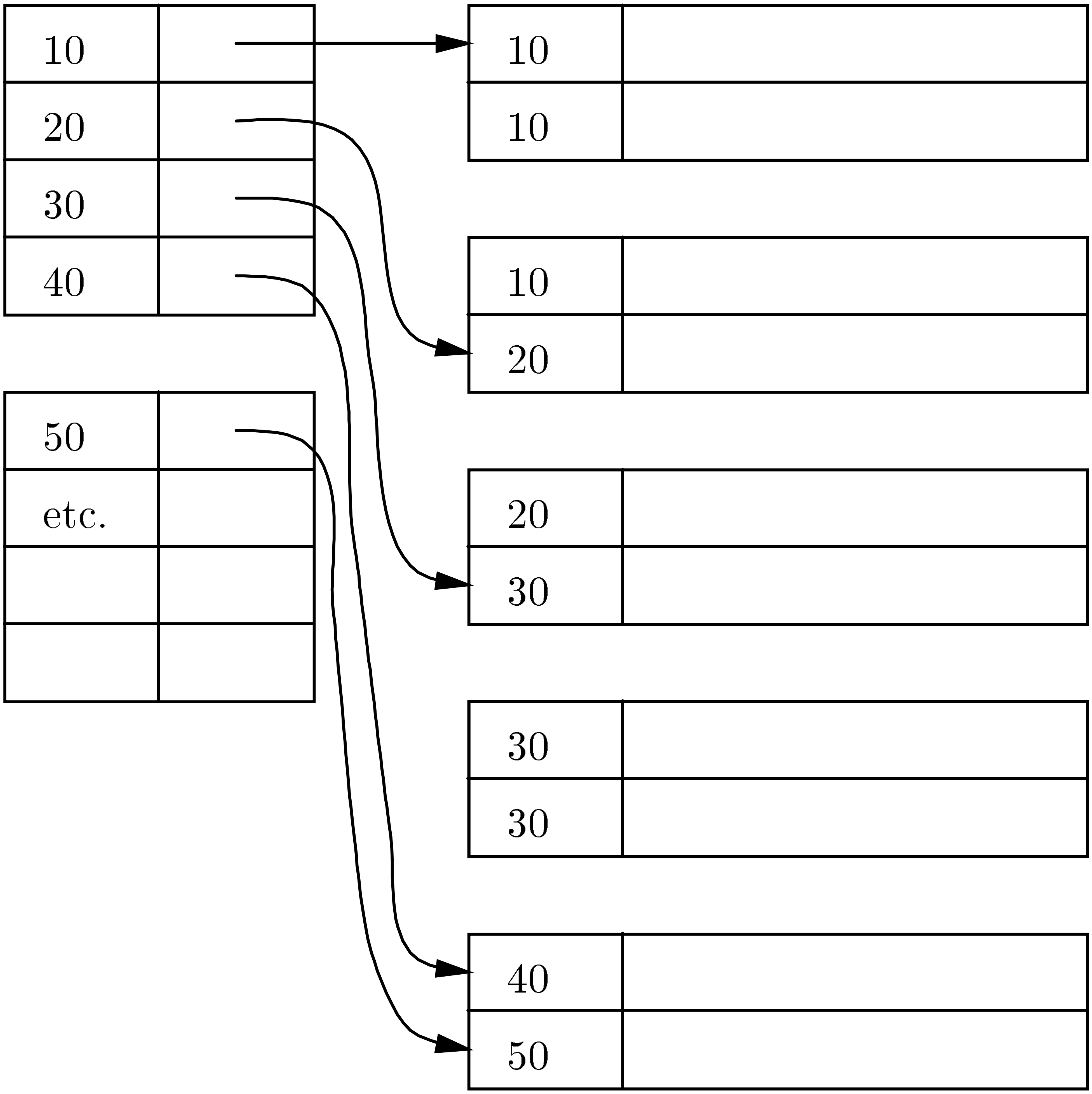
10
20
30
40
50
60
70
80
90
100
110
120
10
20
30
40
50
60
70
80
90
100


10
30
50
70
90
110
130
150
170
190
210
230
10
20
30
40
50
60
70
80
90
100

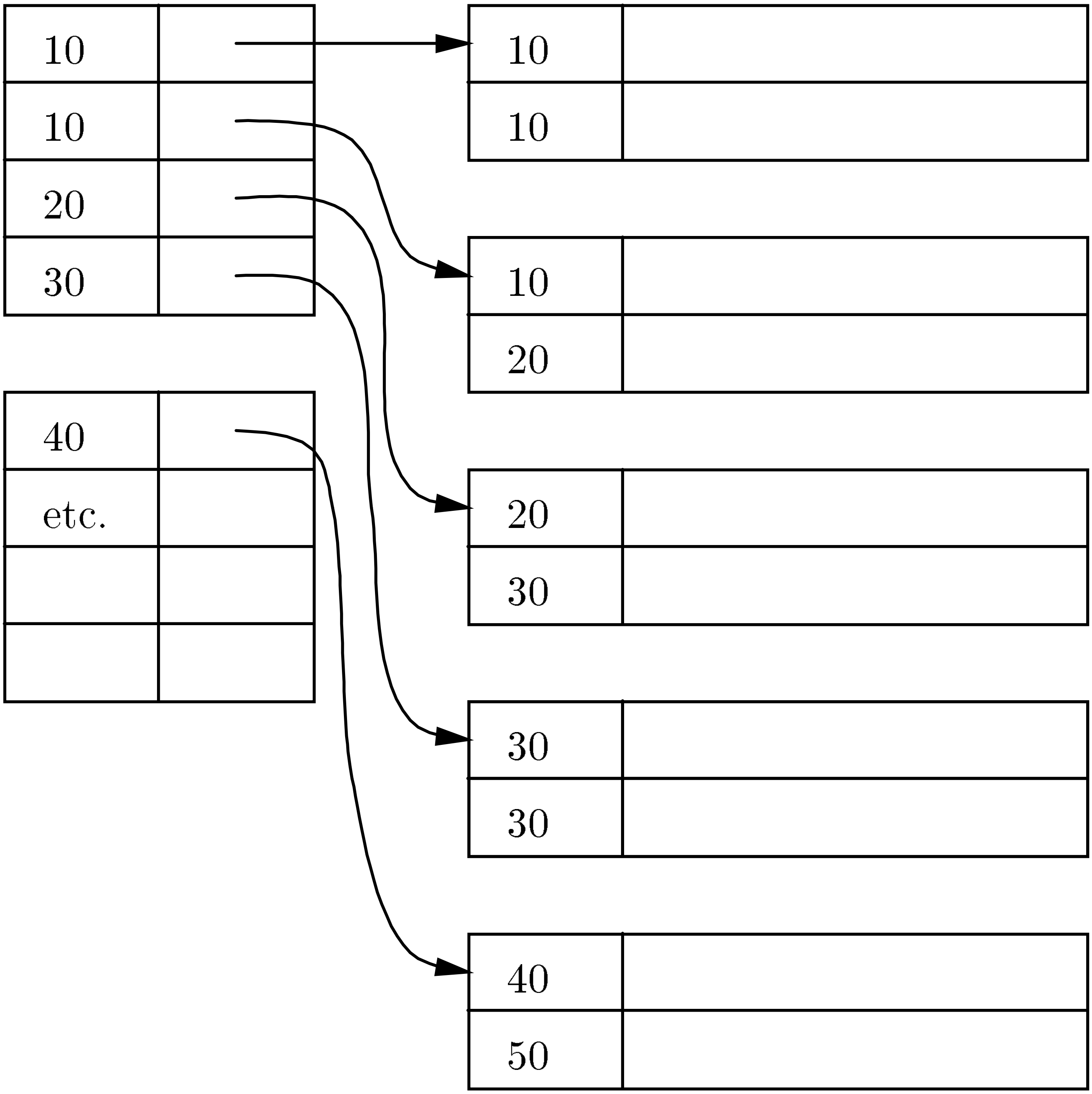
Duplicate Search Keys

- Dense index:
  - a) Duplicate key-pointer pairs.
  - b) Pointers to only first record with given search key.
- Sparse index:
  - a) Pointer to first record of each block.
  - b) Pointer to first *new* key on block (sole key if all are the same).

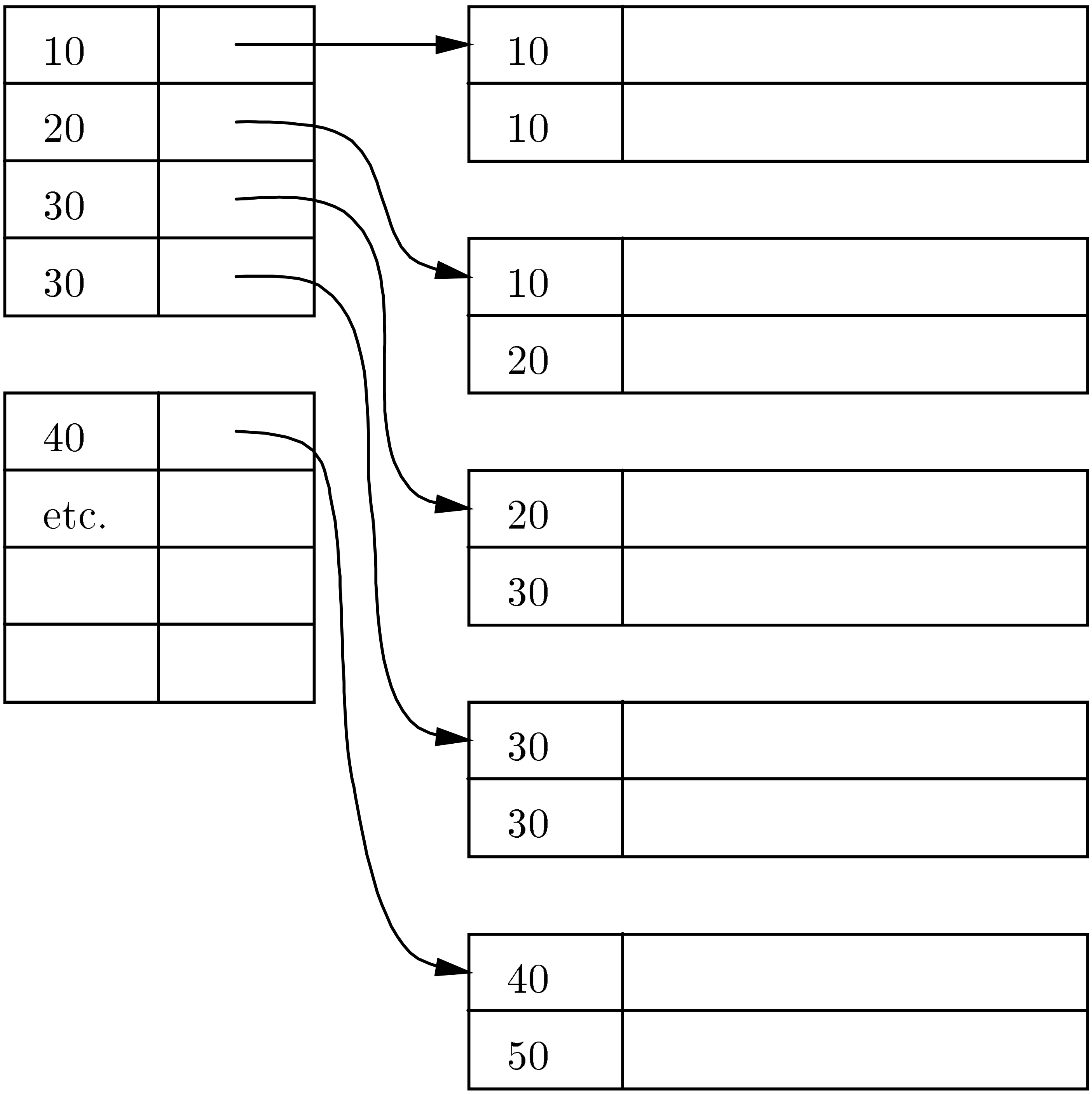
First Key Occurrences Only



Sparse, First Key on Block



Sparse, First New Key



Lookup

Issues: sparse vs. dense? Duplicates? Where are other records with same key (ahead or behind)?

1. Find key in dense index; find greatest key $\leq$ in sparse.

2. Follow pointer.

   a) Dense, no duplicates: just follow.

   b) Dense, duplicates: follow each (pointer per record) or follow and look at successive records (pointer to first with given key).

   c) Sparse, no duplicates: follow to block, examine block.

   d) Sparse, duplicates, key = lowest in block: follow to block, look at block, successive blocks until higher key met, and (if key = desired key) previous block.

   e) Sparse, duplicates, key = lowest new in block: follow to record, search following records of block, and successive blocks until higher key met.

Multiple Levels of Index

A sparse index on a (sparse or dense) index is an option.

- Good chance that 2nd or higher level indexes can be housed in main memory, so no additional disk I/O's.
- Dense higher level indexes make no sense; dense(dense) = same dense index.