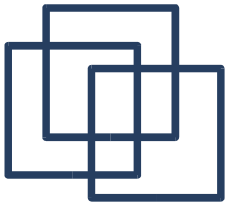
Representing Data Elements

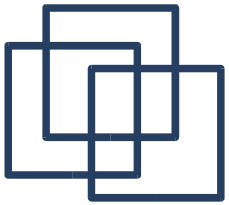
Bogdan Dumitriu

November 24th, 2004



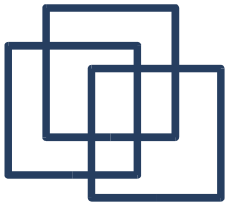
Summary

- Representing data elements
  - Data elements and fields
  - Records
  - Representing block and record addresses
  - Variable-length data and records
  - Record modifications
- Index on sequential files



Topics

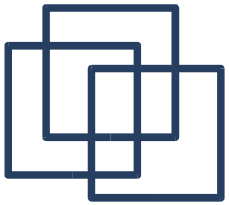
- Representing data elements
  - Data elements and fields
  - Records
  - Representing block and record addresses
  - Variable-length data and records
  - Record modifications
- Index on sequential files



CHAR vs. VARCHAR

	CHAR(N)	**VARCHAR(N)**
book	N	N+1
PostgreSQL	N+4	L+4
MS SQL Serv.	N?	L+2?
MySQL	N	L+1
Oracle	N	L+2?

Note: L – actual length of stored string

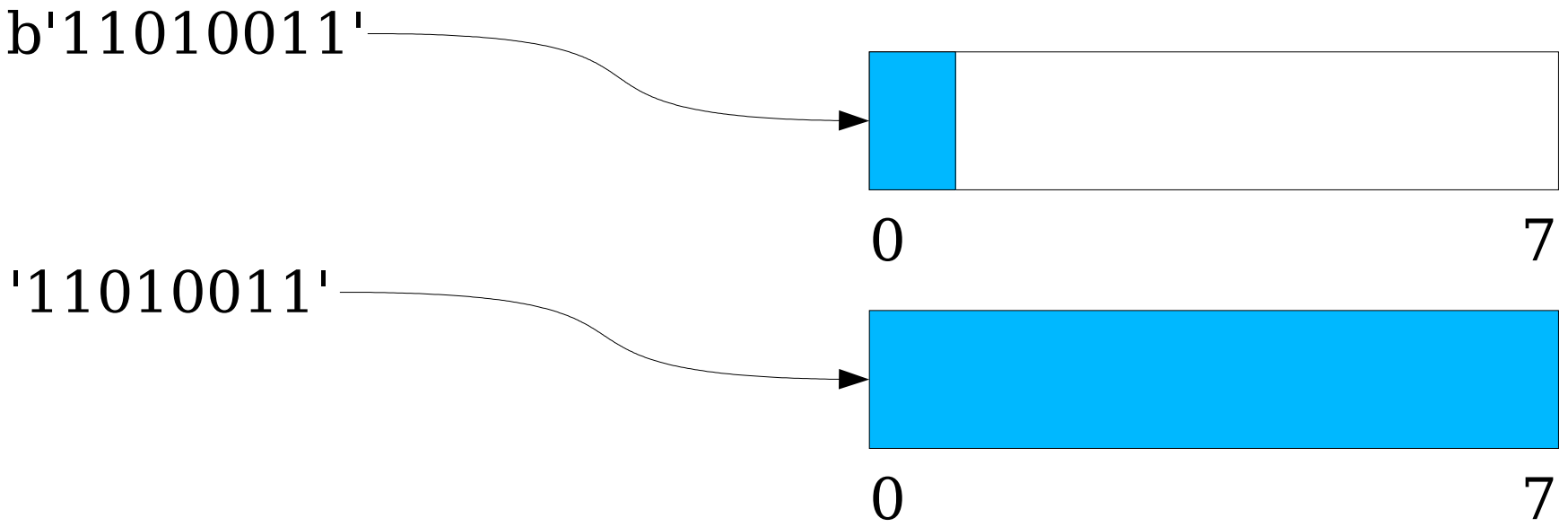


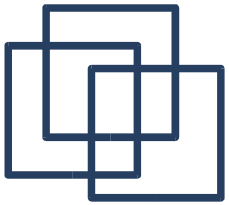
CHAR vs. VARCHAR

- space usage – use VARCHAR
- fast search/sort – use ?
- better management – use CHAR
- minimize corruption – use CHAR
- comparison semantics – varies
- tip: if n < 4 always use CHAR

BIT [VARYING]

- bit logical operators (&, |, <<, etc.)
- additional type checking
- allows more efficient storage:





Topics

- Representing data elements
  - Data elements and fields
  - Records
  - Representing block and record addresses
  - Variable-length data and records
  - Record modifications
- Index on sequential files

Data alignment

instruction word

...	01110110...11	...

32 bits

1011001...01

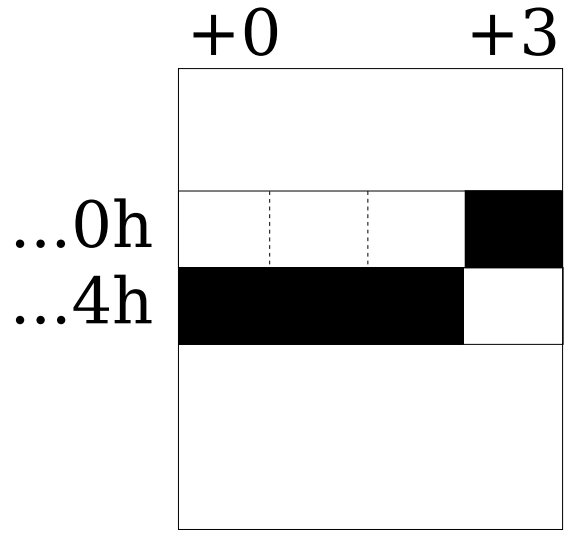
(32 bits)

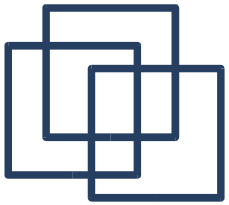
32

<

10110...0100

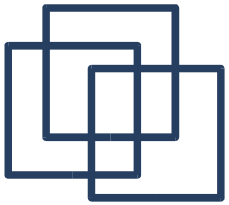
(34 bits)





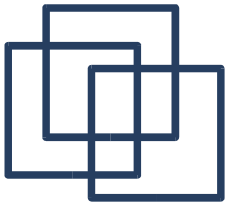
Record schema

- Reasons for record schema
  - accomodates schema changes
  - more types of records in same block
    - that's why we need r.s. pointer in record

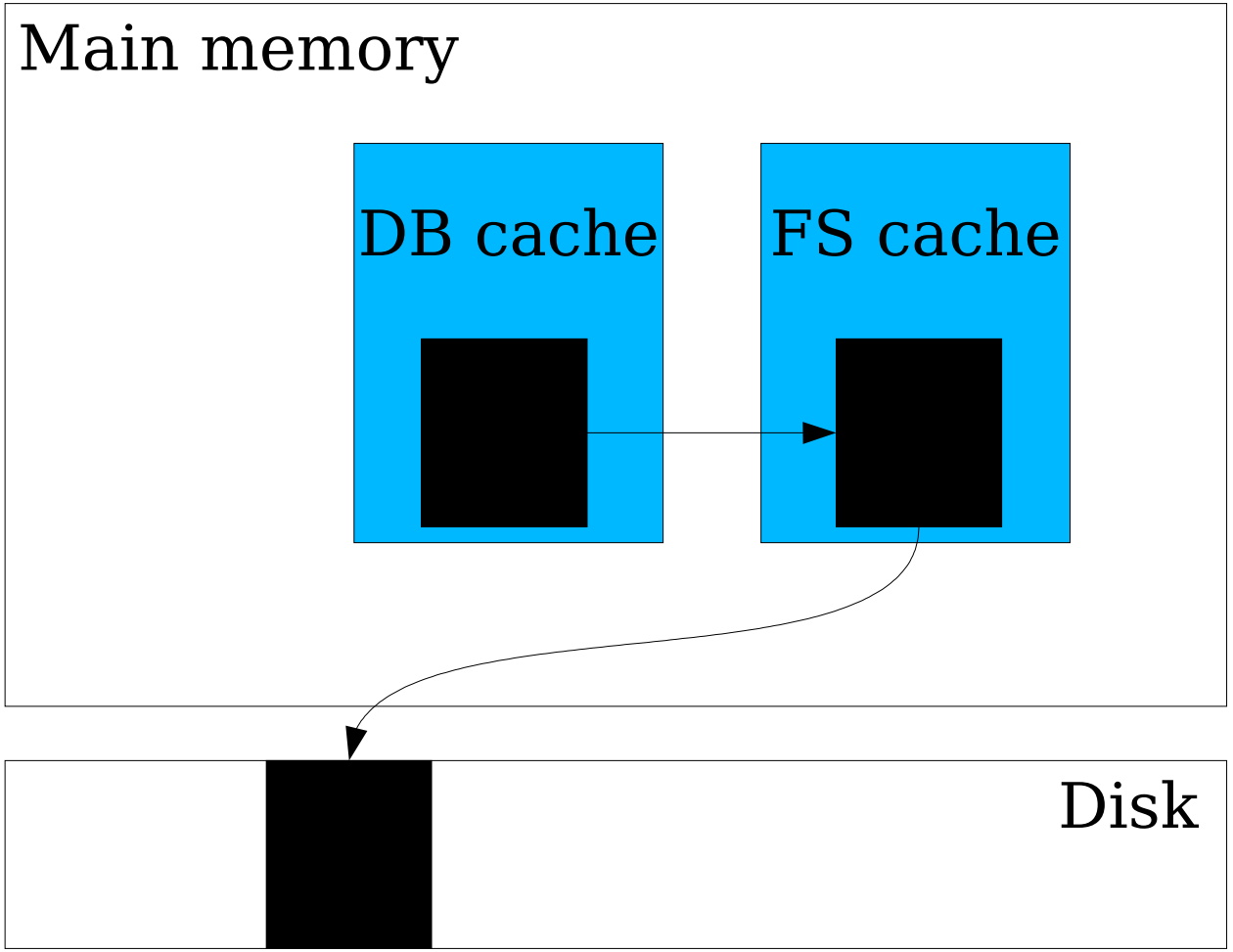


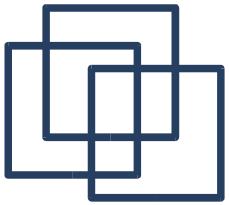
Block size

- File system based data files
  - direct I/O
  - buffered I/O
- Raw I/O data files



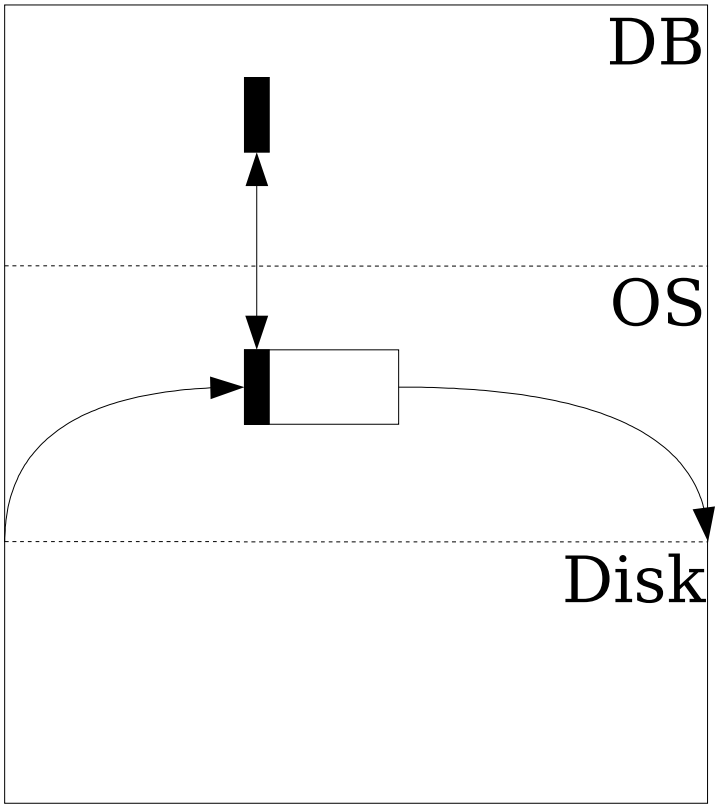
Buffered I/O



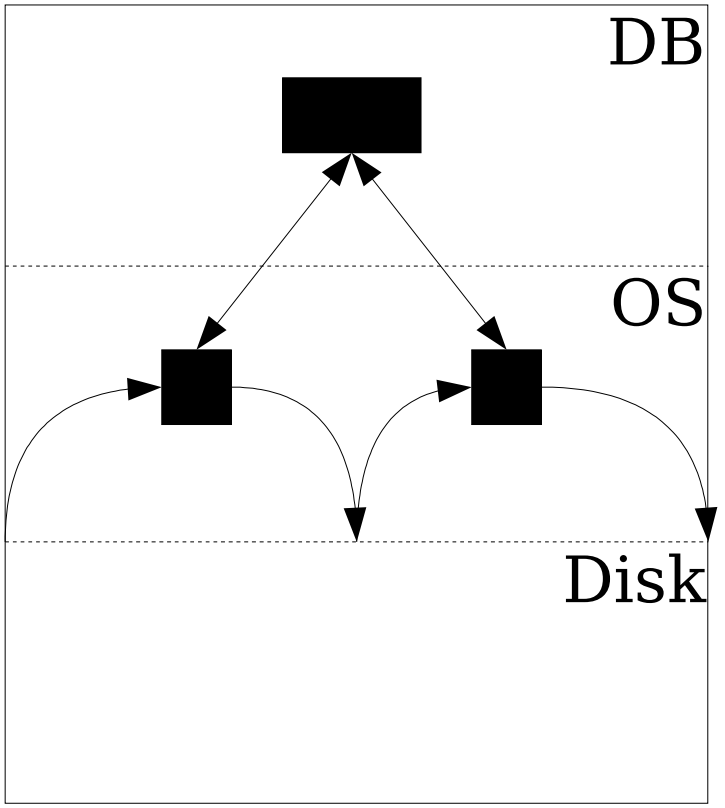


Buffered I/O

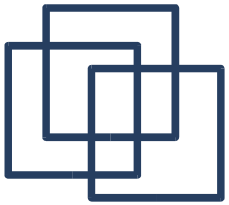
- use FS buffer size = DB block size



partial block writes

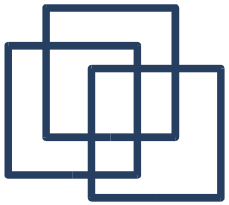


I/O fragmentation



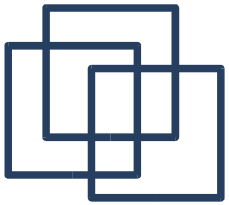
Direct/raw I/O

- Allows a wider selection of block sizes
- Decision based on
  - size of records
  - type of database/table access
  - many others
- General tendency: increase block size



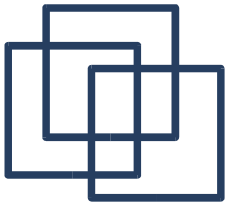
Small block size

- good for small rows
- good for non-sequential access
- reduces block contention
- large space overhead
- bad for large rows



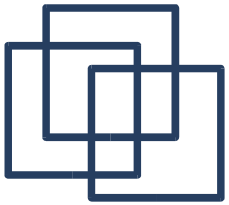
Large block size

- good for large rows (reduces chaining)
- good for sequential access
- wastes buffer cache
- increased block contention
  - bad for indexes



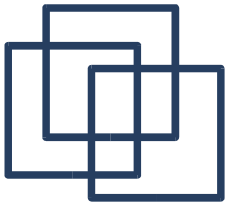
Topics

- Representing data elements
  - Data elements and fields
  - Records
  - Representing block and record addresses
  - Variable-length data and records
  - Record modifications
- Index on sequential files



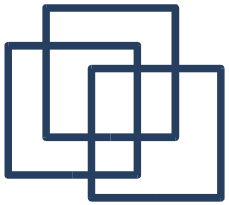
DB Addresses

- Physical address
  - no translation
  - longer
- Logical address
  - translation needed (using map table)
  - shorter
  - allows flexibility

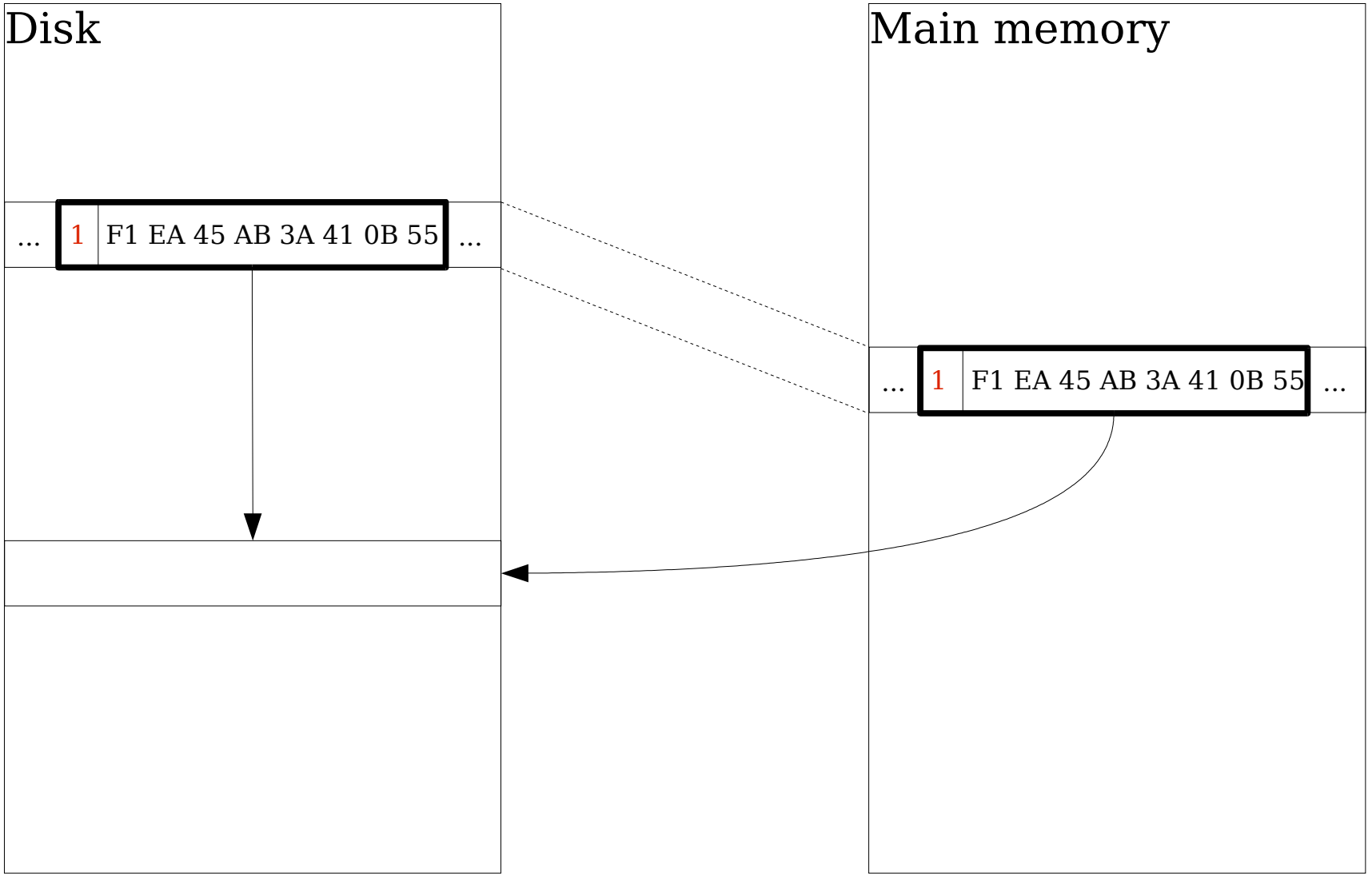


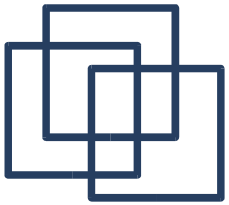
Structured addresses

- Disadvantages of offset tables
  - waste of space (4KB block => 10 bits/rec)
  - additional level of indirection
- Header format
  - depends on addressing scheme
  - key-based – just use a count
  - offset table – just use the offset table
  - …

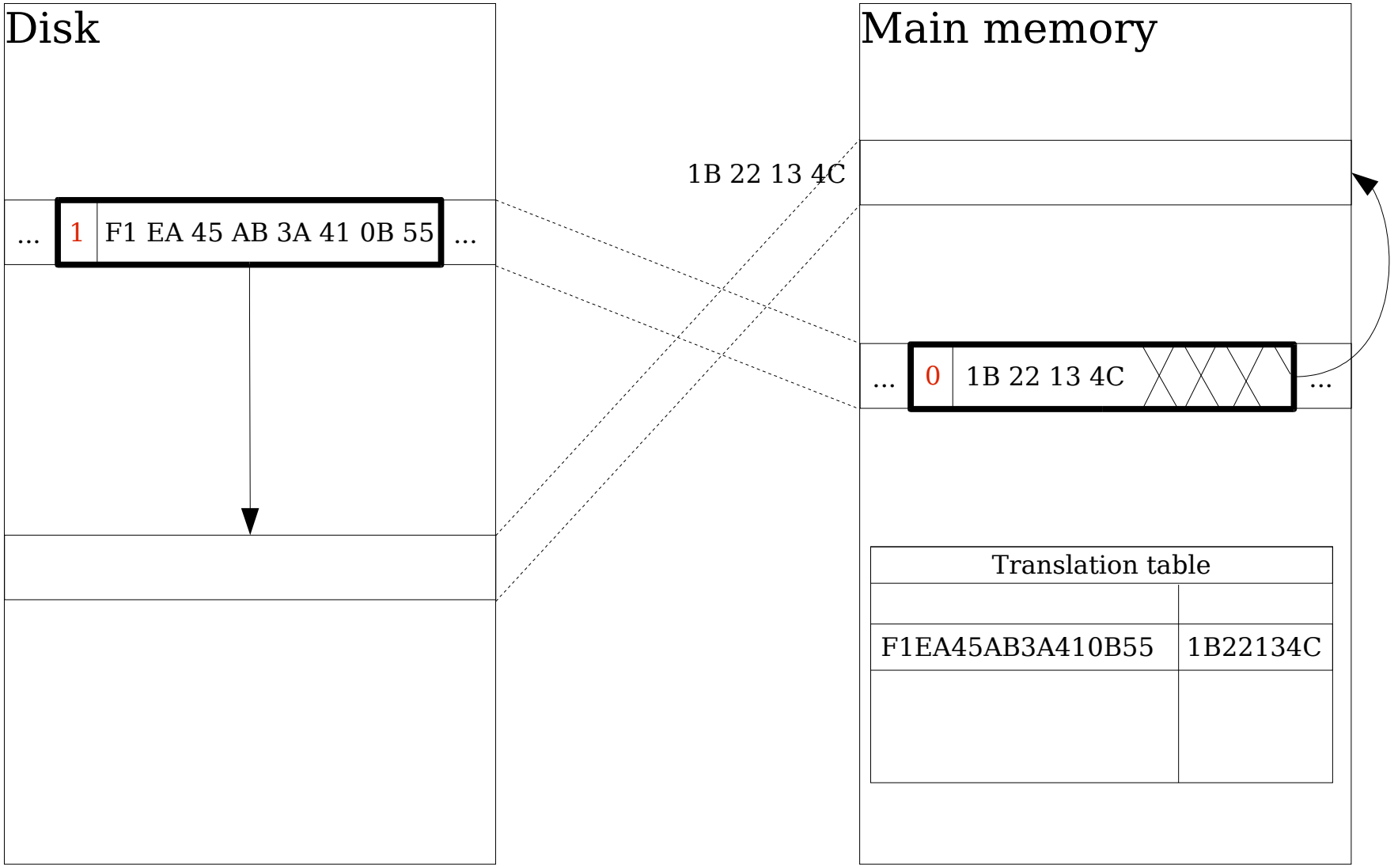


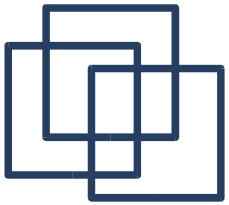
Pointer swizzling





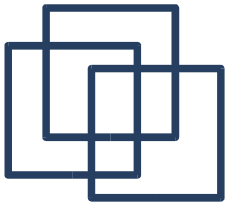
Pointer swizzling





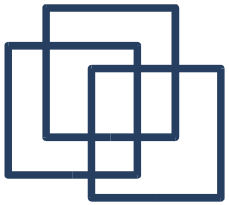
Swizzling techniques

- Automatic
  - block -> memory
  - all addressable items -> transl. table
  - swizzle all possible pointers in block on load
- On demand
  - block -> memory
  - all addressable items -> transl. table
  - swizzle pointers in block on demand



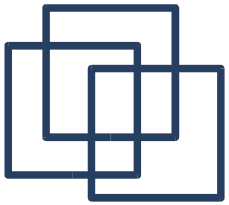
Swizzling

- Efficiency depends on app. profile
  - follow most pointers => use automatic
  - follow just a few pointers => use on demand
  - adaptable swizzling
- Why swizzle?
  - no swizzling => use TT every time
  - swizzling => use TT once (maybe twice)



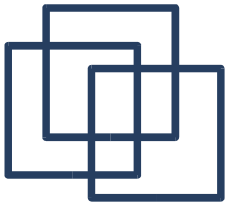
Swizzling

- Virtual memory approach
  - easier management
  - (very) expensive
  - enhancement: swizzle all pointers in block
  - enhancement: persist swizzled pointers
- ''Software'' approach
  - more difficult management
  - less costly



Unswizzling

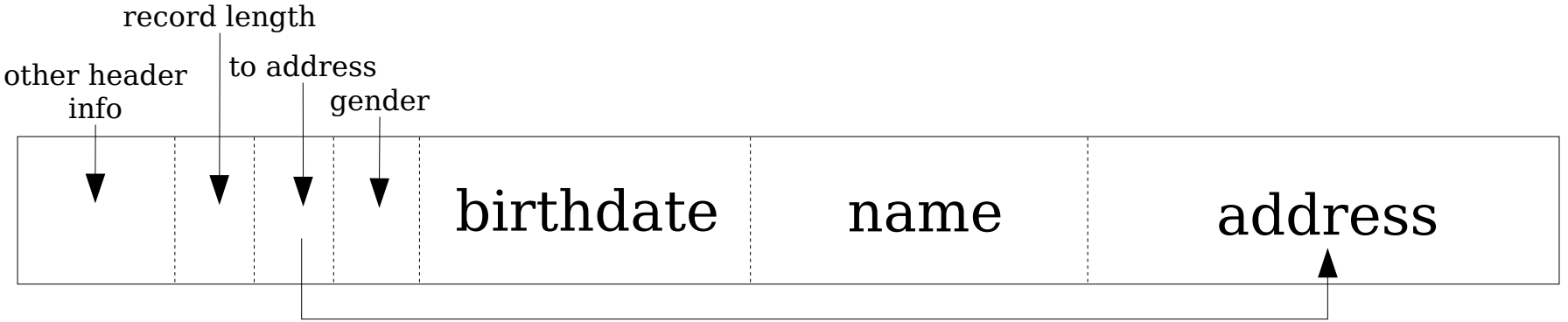
- Ensuring pointer correspondence
  - no change – address stays the same
  - changes in records
    - unswizzle all pointers
    - use indirect swizzling
- Efficient unswizzling
  - use a hash table
  - use an index or linked list



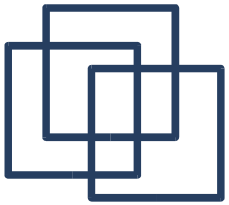
Topics

- Representing data elements
  - Data elements and fields
  - Records
  - Representing block and record addresses
  - Variable-length data and records
  - Record modifications
- Index on sequential files

NULLs

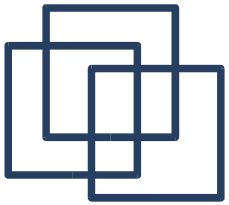


- Use info on...
  - fixed length part of record
  - first non-NULL pointer
  - record length
- ...or just add a pointer for name field



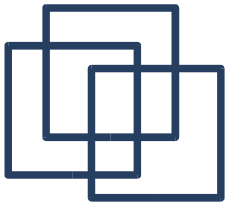
Efficiency

- Variable-length fields
  - are always less efficient
  - sometimes save space
- Only solution
  - use fixed-length fields



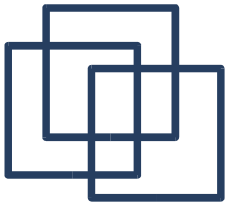
Tagged fields

- Useful in relational DBs:
  - information-integration apps
  - records with flexible schema
  - main reason: save space



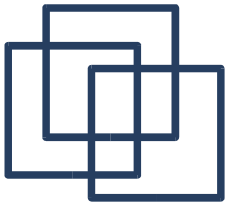
Why use BLOBs?

- performance is not the only issue!
- may need ACID properties
- may need portability
- may need access rights
- easier management
- may even want to index BLOB field
- performance might not be affected



Why use files?

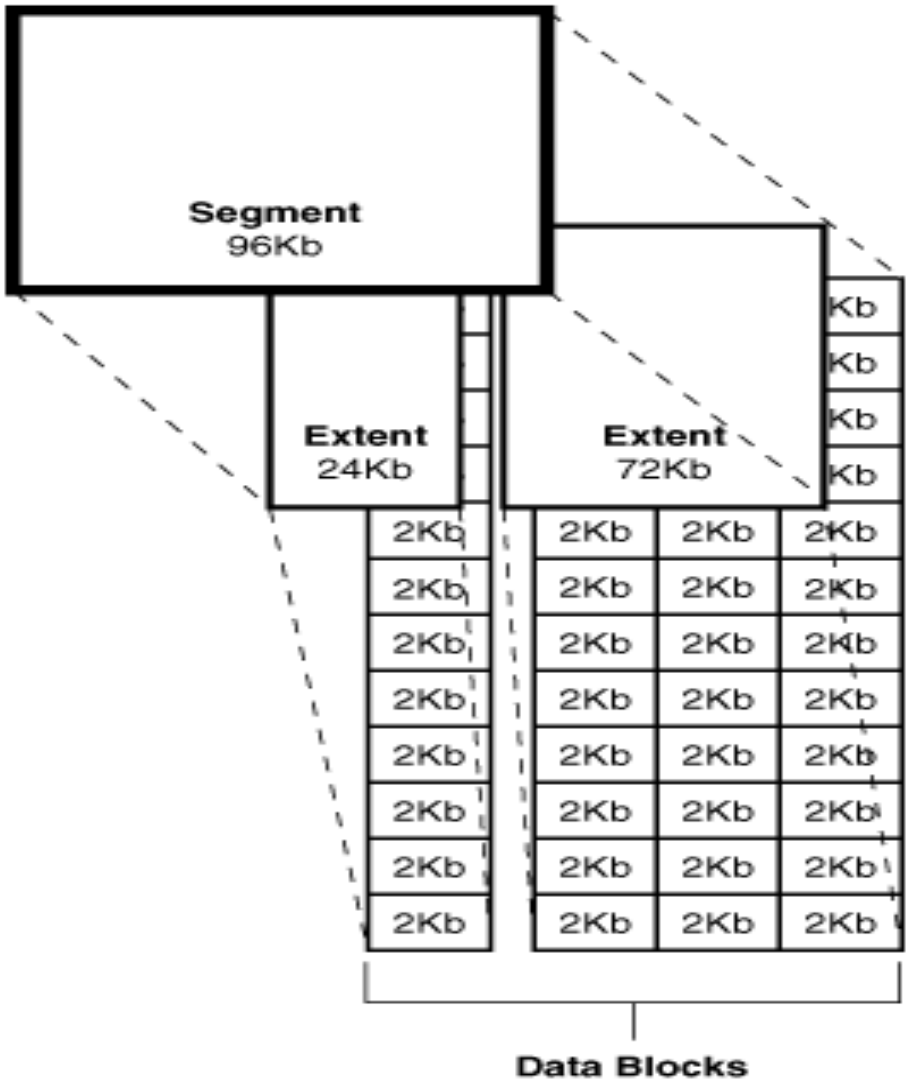
- could be more efficient
- BLOBs can make DB's huge
- easier manipulation of data
- allows caching (with web pages)



Topics

- Representing data elements
  - Data elements and fields
  - Records
  - Representing block and record addresses
  - Variable-length data and records
  - Record modifications
- Index on sequential files

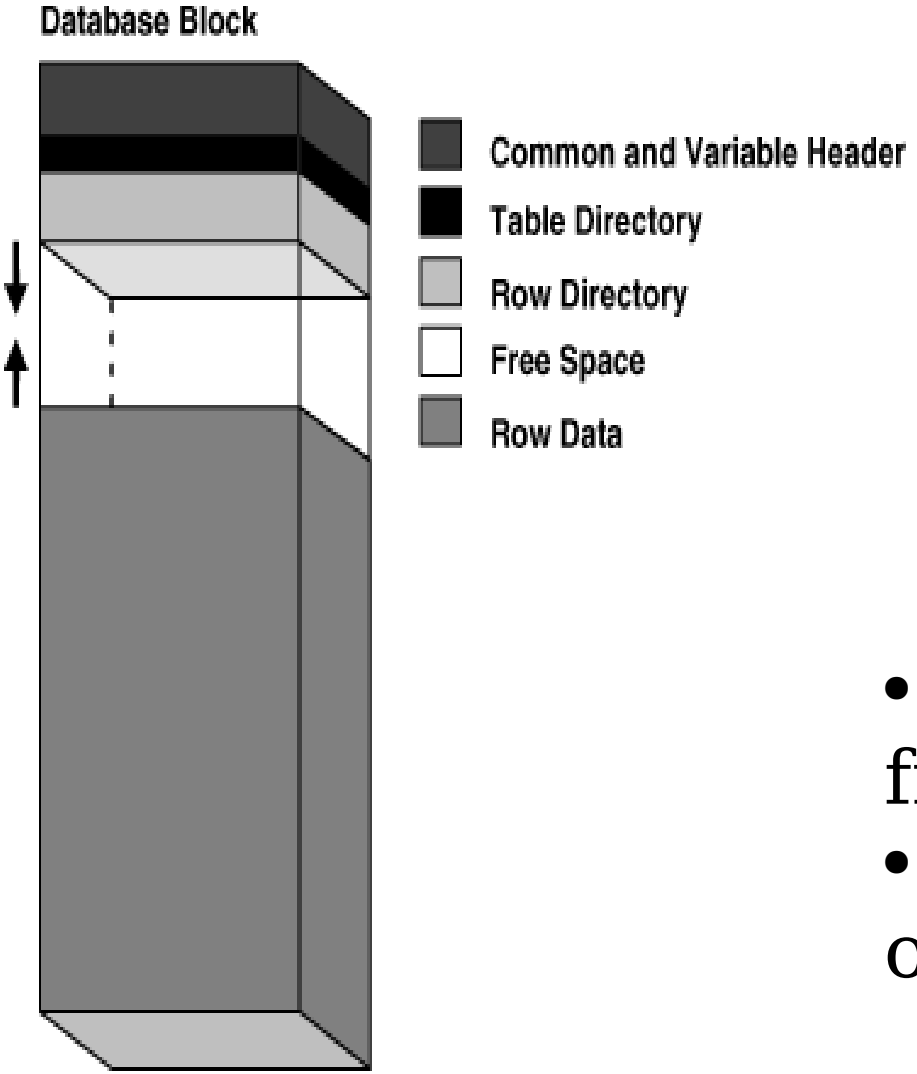
Free space mgmt.



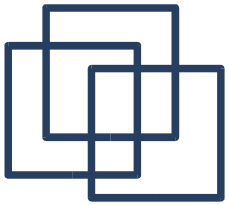
Oracle logical units:

- block
  - smallest unit that can be used
- extent
  - contiguos blocks
- segment
  - (non-)contiguous extents

Free space mgmt.



- DELETE's & UPDATE's free space
- free space compressed only when needed

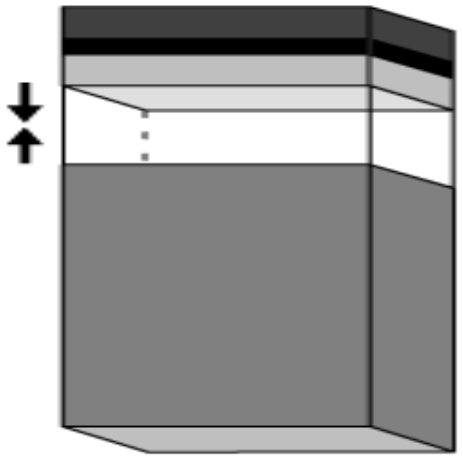


Free space mgmt.

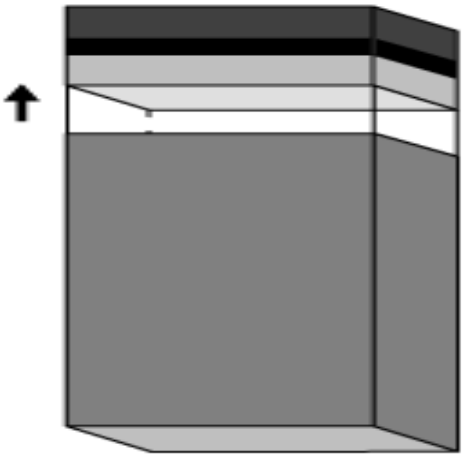
- row too large to fit in one block
  => row chaining
- row is updated, block is already full
  => row migration
- I/O performance decreases

Free space mgmt.

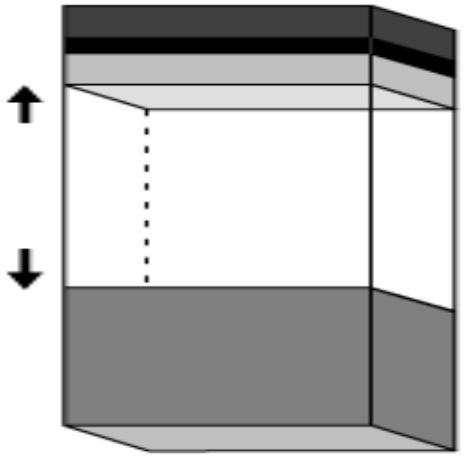
Data Block
PCTFREE = 20, PCTUSED = 40



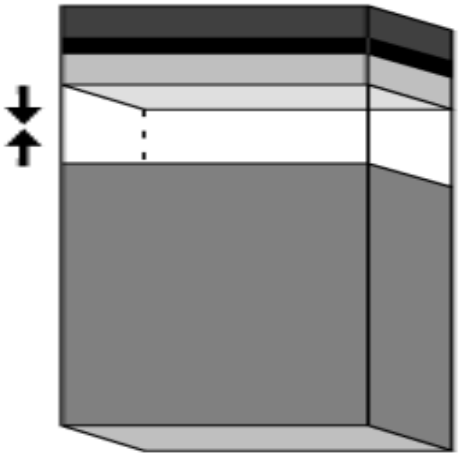
1 Rows are inserted up to 80% only, because PCTFREE specifies that 20% of the block must remain open for updates of existing rows.



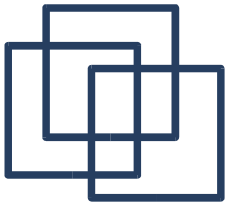
2 Updates to exisiting rows use the free space reserved in the block. No new rows can be inserted into the block until the amount of used space is 39% or less.



3 After the amount of used space falls below 40%, new rows can again be inserted into this block.

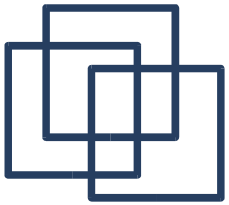


4 Rows are inserted up to 80% only, because PCTFREE specifies that 20% of the block must remain open for updates of existing rows. This cycle continues . . .



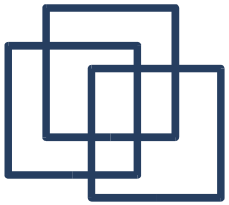
Free space mgmt.

- Find space on ''nearby'' block
  - could ''save the day'' in some cases
  - cannot be used alone
- Create an overflow block
  - easier to index
  - problems during retrieval
- Solution: page (block) split



Tombstones

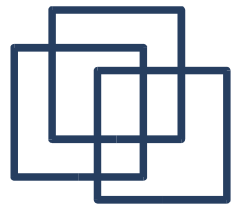
- Tombstones vs. NULL
  - tombstone is just a marker
  - NULL is a marker as well
  - conclusion: they are equivalent



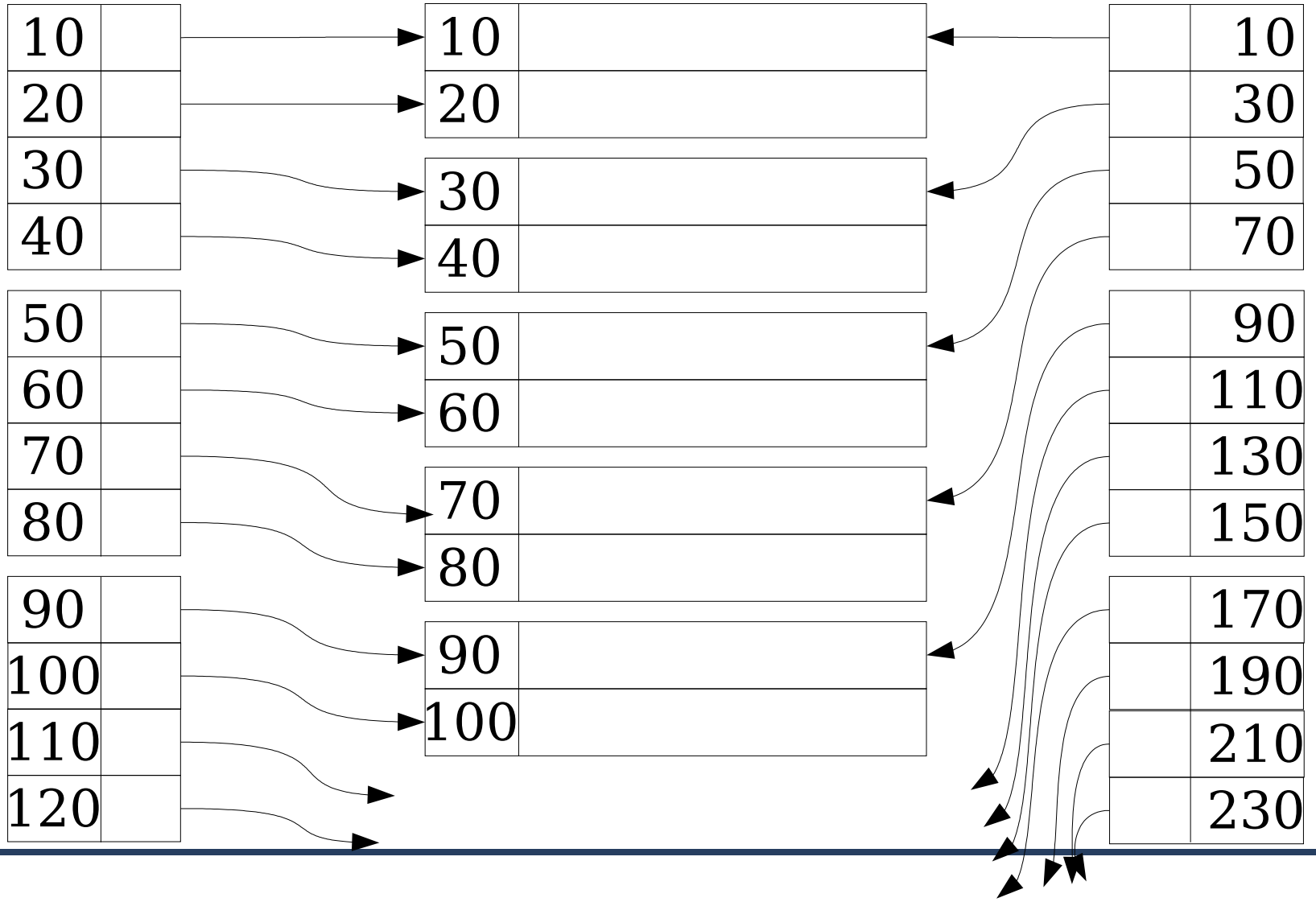
Topics

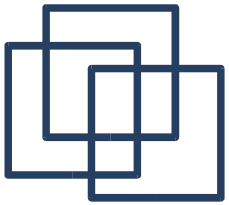
- Representing data elements
  - Data elements and fields
  - Records
  - Representing block and record addresses
  - Variable-length data and records
  - Record modifications

- Index on sequential files

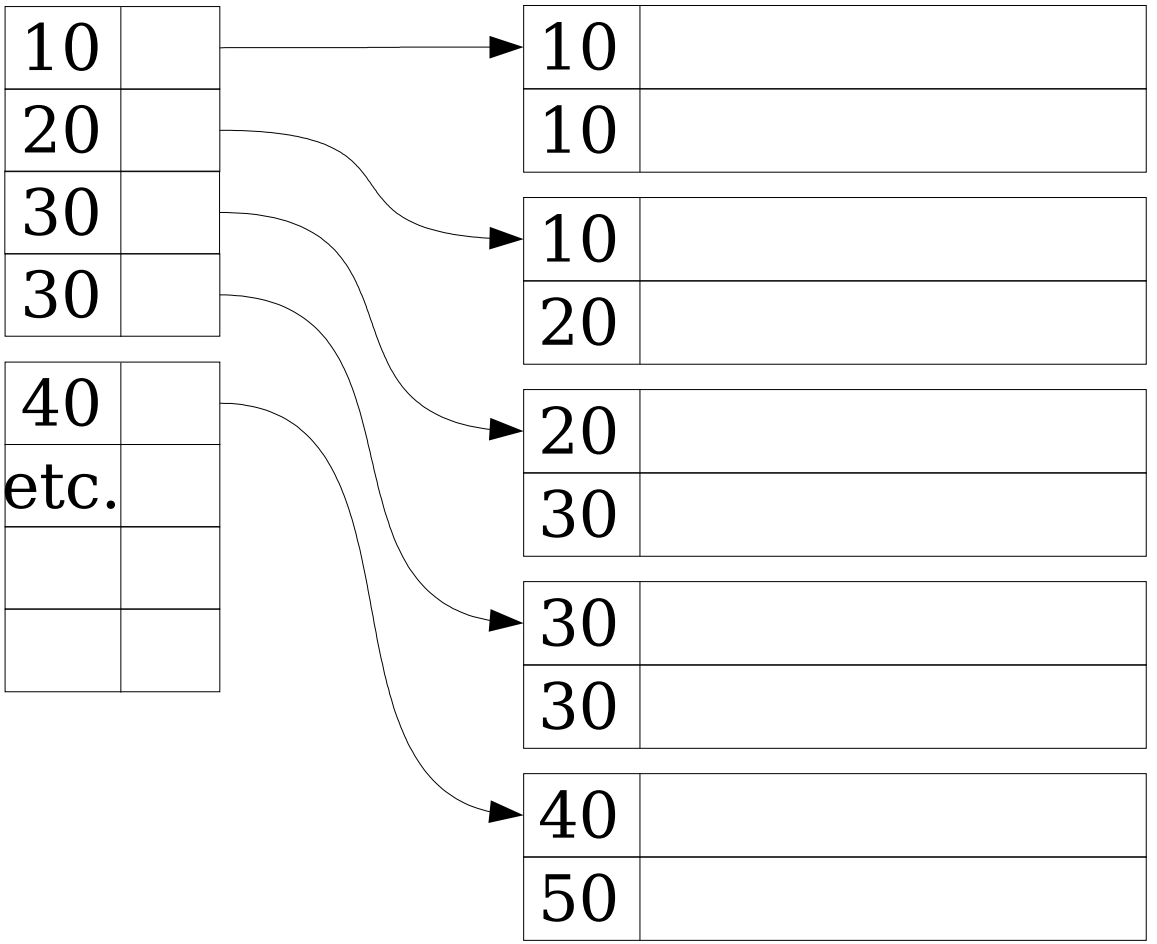


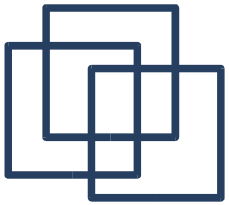
Dense/sparse indexes





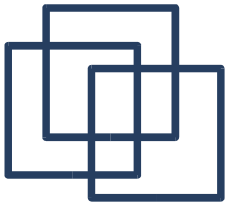
Example 4.8





Managing sparse ind.

- Insert record
  - if key is smallest, update
  - else, leave unchanged
- Delete record
  - if smallest key deleted, update
  - else, leave unchanges
- Slide record
  - if smallest key in either block changes, up.



Conclusions

- None really, except that 36h is way too little time for 31 questions...
- Any questions (submitted or not) left unanswered?