Electronics and
Computer Science

UNIVERSITY OF
Southampton

Data Storage

COMP3211 Advanced Databases

Dr Nicholas Gibbins - nmg@ecs.soton.ac.uk
2016-2017

Overview

- Storage Organisation
- Secondary storage
- Buffer management
- The Five-Minute Rule
- Disk Organisation
  - Data Items
  - Records
  - Blocks

Storage Organisation

The Memory Hierarchy: Cache

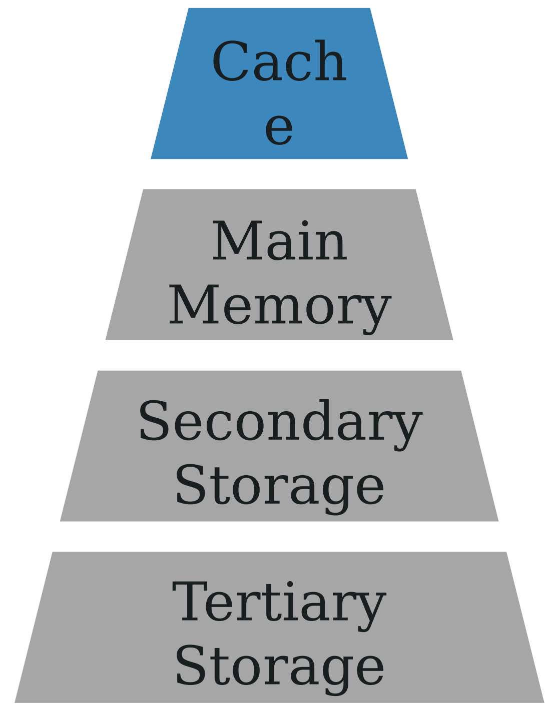
Volatile storage

Very fast, very expensive, limited capacity

Hierarchical

Typical capacities and access times:

- Registers – $\sim 10^1$ bytes, 1 cycle
- L1 – $\sim 10^4$ bytes, <5 cycles
- L2 – $\sim 10^5$ bytes, 5-10 cycles



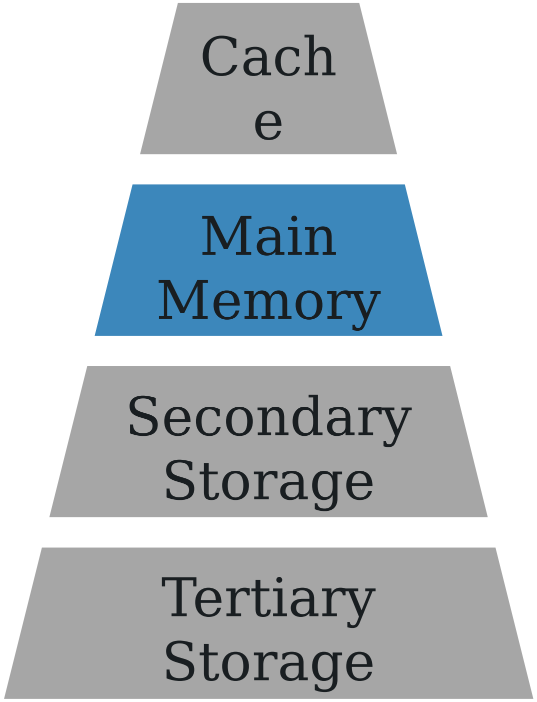
The Memory Hierarchy: Main Memory

Volatile storage

Fast, affordable, medium capacity

Typical capacity: 10^9-10^{10} bytes

Typical access time: 10^{-8} s (20-30 cycles)



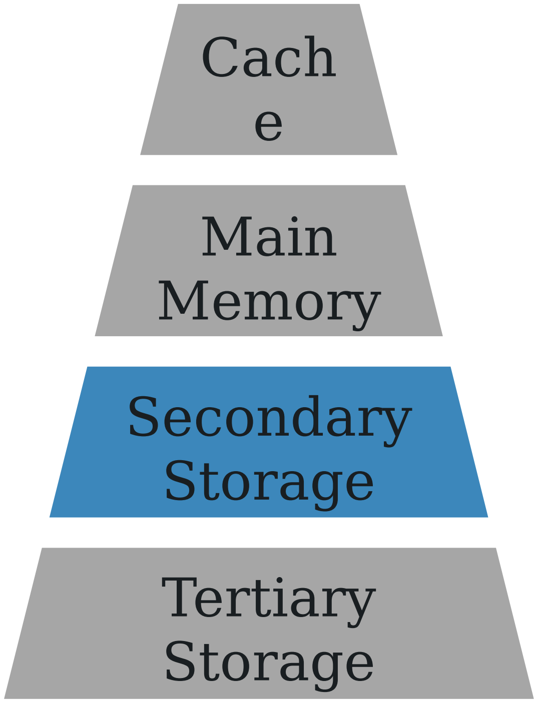
The Memory Hierarchy: Secondary Storage

Non-volatile storage

Slow, cheap, large capacity

Typical capacity: 10^{11}-10^{12} bytes

Typical access time: 10^{-3} s (10^{6} cycles)



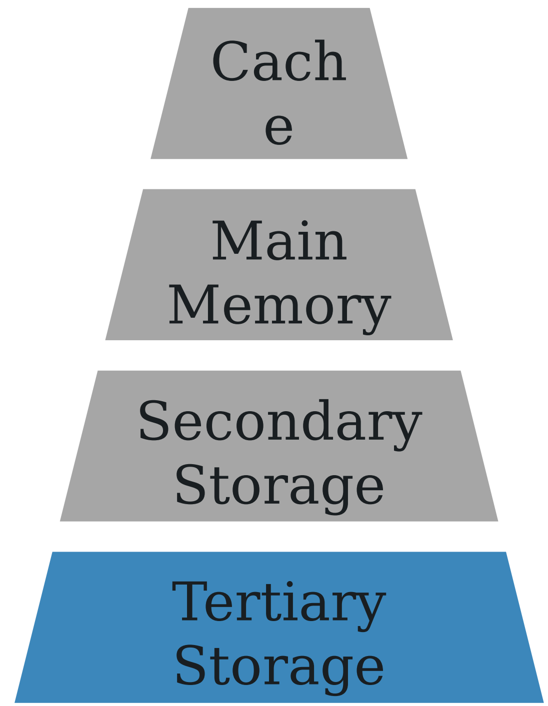
The Memory Hierarchy: Tertiary Storage

Non-volatile storage

Very slow, very cheap, very large capacity

Typical capacity: 10^{13}-10^{17} bytes

Typical access time: 10^{1}-10^{2} s



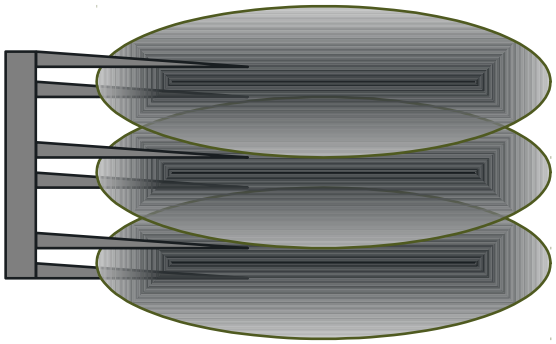
Secondary Storage

Hard Disk Drives

Typical secondary storage medium for databases

Terms:

- Platter, surface, head, actuator, cylinder, track, sector, block, gap



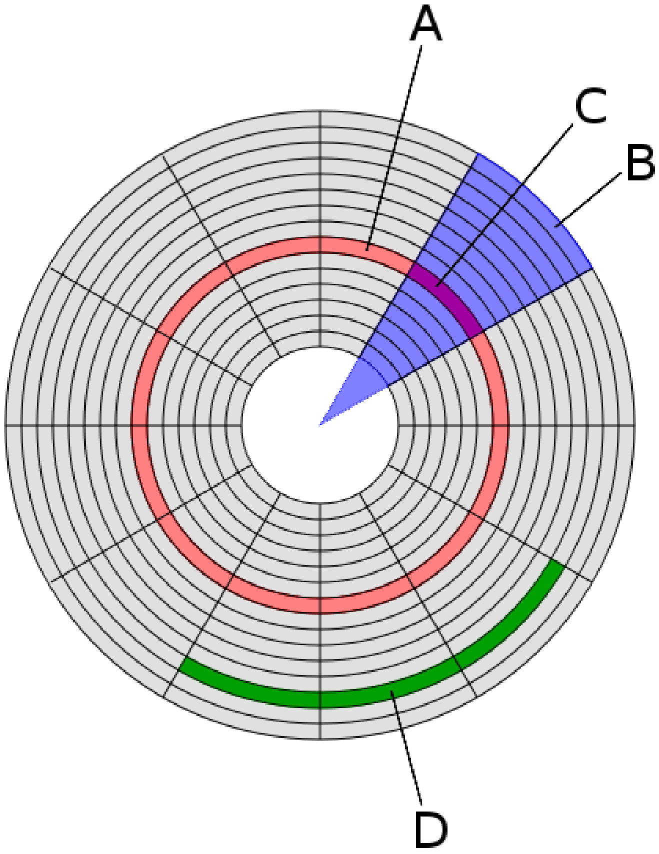
Disk Structure

A – track

B – geometrical sector

C – track sector

D – cluster



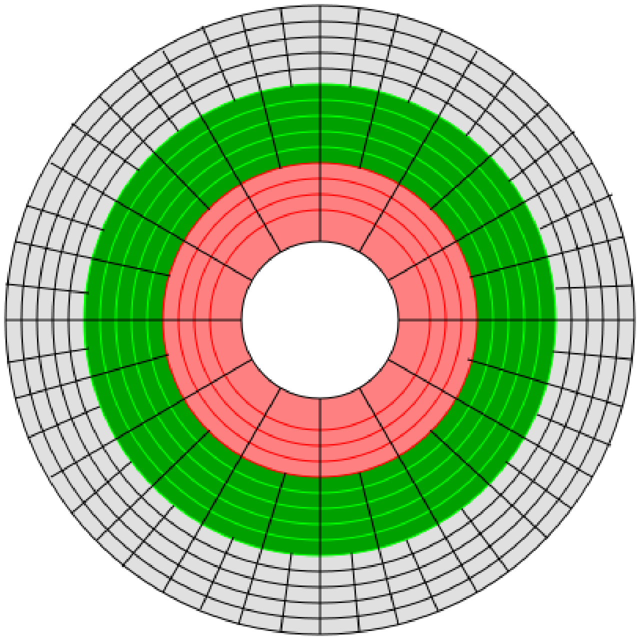
Zone Bit Recording

Tracks closer to the disc edge are longer than those closer to the axis

- Bit densities vary in order to ensure a constant number of bits per sector

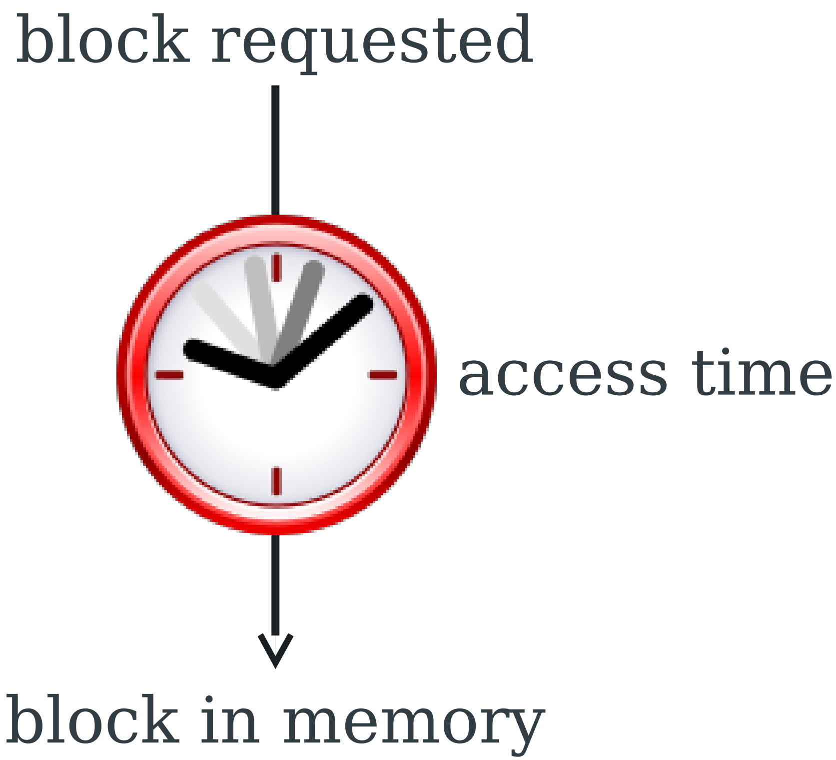
Instead, we can vary the number of sectors per track (depending on track location)

- Improves overall storage density



Disk Access Time: Reading

Access Time = Seek Time +
Rotational Delay +
Transfer Time



Seek Time

Time taken for head assembly to move to a given track

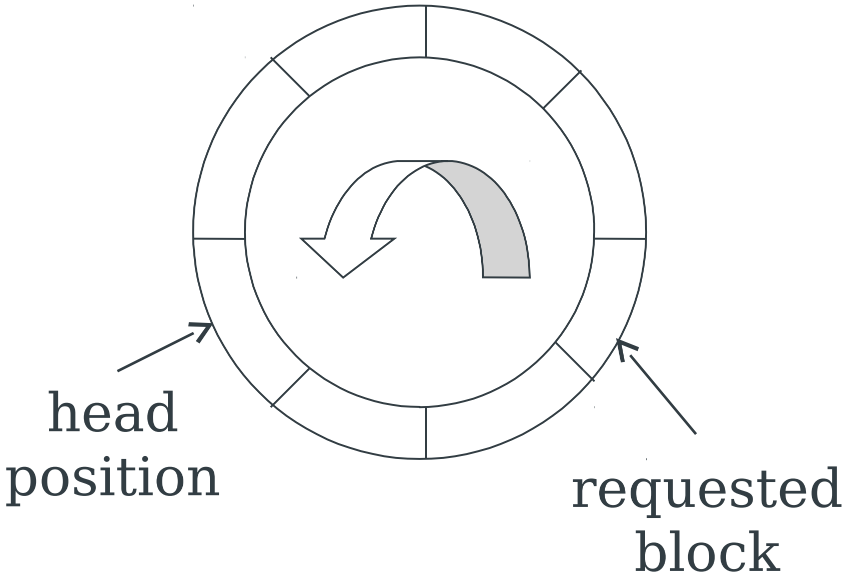
Average seek time range:

- 4ms for high end drives
- 15ms for mobile devices

Rotational Delay (Latency)

Average delay = time for 0.5 rev

Rotational speed [rpm]	Average delay [ms]
4,200	7.14
5,400	5.56
7,200	4.17
10,000	3.00
15,000	2.00



Transfer Time

Transfer rate ranges from:

- up to 1000 Mbit/sec
- 432 Mbit/sec 12x Blu-Ray disk
- 1.23 Mbits/sec 1x CD
- for SSDs, limited by interface
  e.g., SATA 3000 Mbit/s

Transfer time = block size / transfer rate

Sequential Access

So far, random access - what about reading “next” block?

Access time = (block size / transfer rate) + negligible costs

Negligible costs:

- skip inter-block gap
- switch track (within same cylinder)
- switch to adjacent cylinder occasionally

- In general, sequential i/o is much less expensive than random i/o

Disk Access Time: Writing

Costs similar to those for reading, unless we wish to verify data

Verifying requires that we read the block we've just written

Access Time = Seek Time +
Rotational Delay (1/2 rotation) +
Transfer Time (for writing) +
Rotational Delay (full rotation) +
Transfer Time (for verifying)

Disk Access Time: Modifying

1. Read Block
2. Modify in Memory
3. Write Block
4. Verify Block (optional)

Disk Access Time: Modifying

Access Time = Seek Time +
Rotational Delay (1/2 rotation) +
Transfer Time (for reading) +
Rotational Delay (full rotation) +
Transfer Time (for writing) +
[Rotational Delay (full rotation) +
Transfer Time (for verifying)]

Block Addressing

Cylinder-head-sector

- Physical location of data on disk
- ZBR causes problems (sectors vary by tracks)

Logical Block Addressing

- Blocks located by integer index
- HDD firmware maps LBA addresses to physical locations on disk
- Allows remapping of bad blocks

Block Size Selection?

The size of blocks affects I/O efficiency:

Big blocks reduce the costs of access

- Fewer seeks (seek time + rotational delay) for the same amount of data

Big blocks also increase the amount of irrelevant data read

- If you're trying to read a single record in a block, larger blocks force you to read more data

But what about Solid State Drives?

Solid State Drives

- Typically based on NAND flash memory
- Considerably more expensive than HDD (~8-9x)
- Typically smaller maximum size than HDD (~1-2TB)
- Considerably higher I/O performance
- Asymmetric read/write performance (writes are slower)
- Limited number of program-erase cycles (~100,000)

HDD versus SSD

Random I/Os per second (IOPS) = 1/ (seek + latency + transfer)

	HDD *	**SSD ****
Random Read IOPS	125-150 IOPS	~50,000 IOPS
Random Write IOPS	125-150 IOPS	~40,000 IOPS

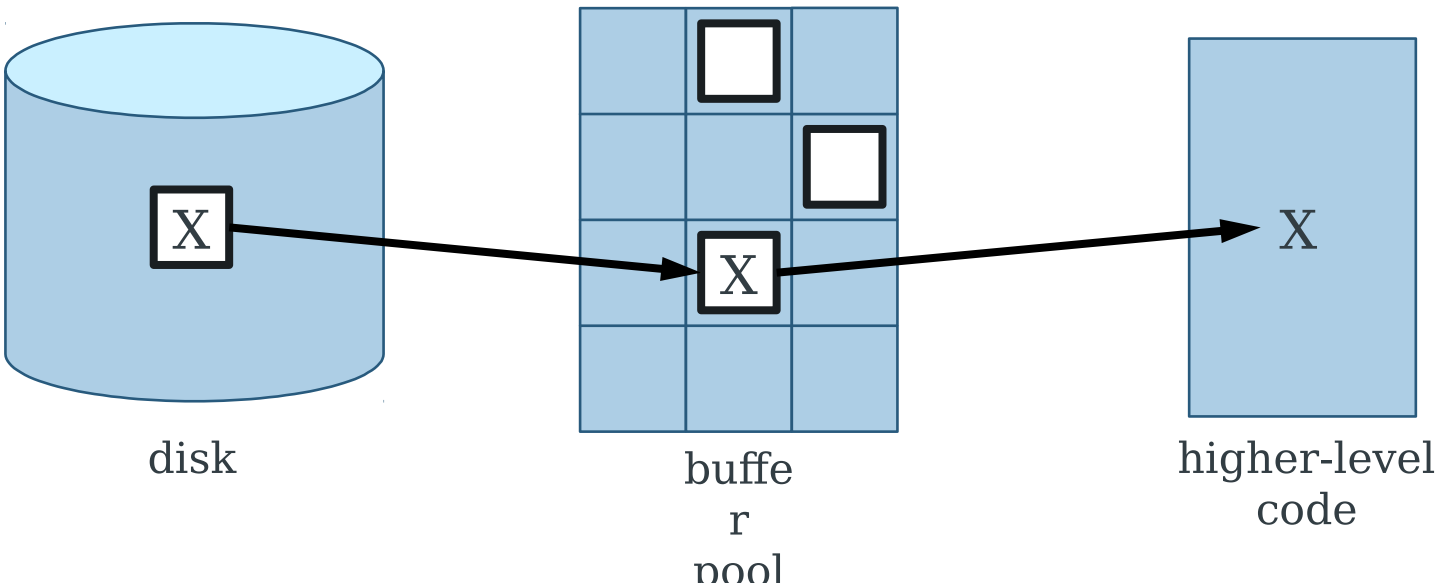
* Assumes 10,000 rpm HDD with SATA 3Gb/s interface
** OCZ 480GB Vertex 3 (c. 2012) with SATA 6Gb/s interface

Buffer Management

The Buffer Pool

Far more blocks of secondary storage than space in main memory – need to be selective about what's kept in memory

Buffer pool organised into frames (size of database block, plus metadata)



Buffer Metadata

Each frame in the buffer pool has:

- a *pin count* (number of current users of the block in that frame)
- a *dirty flag* (1 if the copy in the buffer has been changed, 0 otherwise)
- an *access time* (optional – used for LRU replacement)
- a *loading time* (optional – used for FIFO replacement)
- a *clock flag* (optional – used for Clock replacement)

Requesting a Block

```
if   buffer pool already has a frame containing the block
then increment pin count ("pin the block")
else if there is an empty frame
     then read block into frame and set pin count to 1
     else choose a frame to be replaced
          if dirty bit on the replacement frame is set
          then write block in replacement frame to disk
          endif
          read requested block into replacement frame
     endif
endif
```

Buffer Replacement Strategies

A frame will not be selected for replacement until its pin count is 0

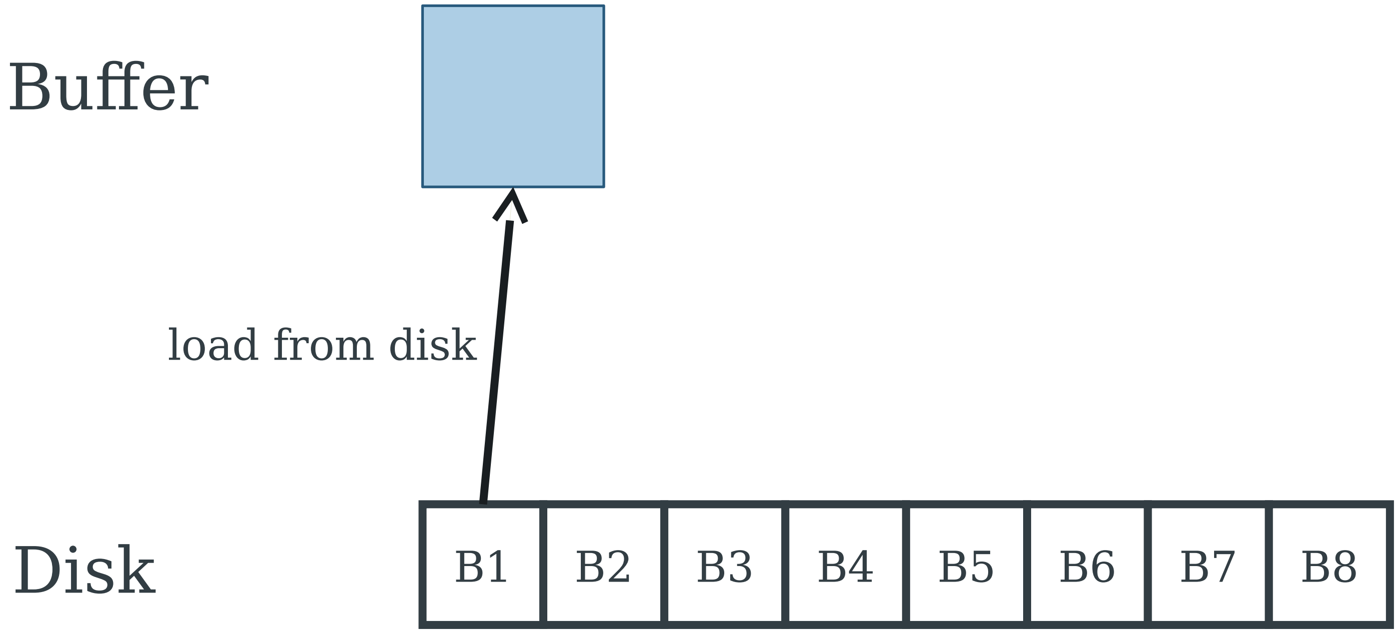
If there's more than one frame with a pin count of 0, use a *replacement strategy* to choose the frame to be replaced

- Least Recently Used (LRU)
  Select the frame with the oldest access time
- First In First Out (FIFO)
  Select the frame with the oldest loading time
- Clock
  Approximation of LRU – cycle through each buffer in turn, if a buffer hasn't been accessed in a full cycle then mark it for replacement

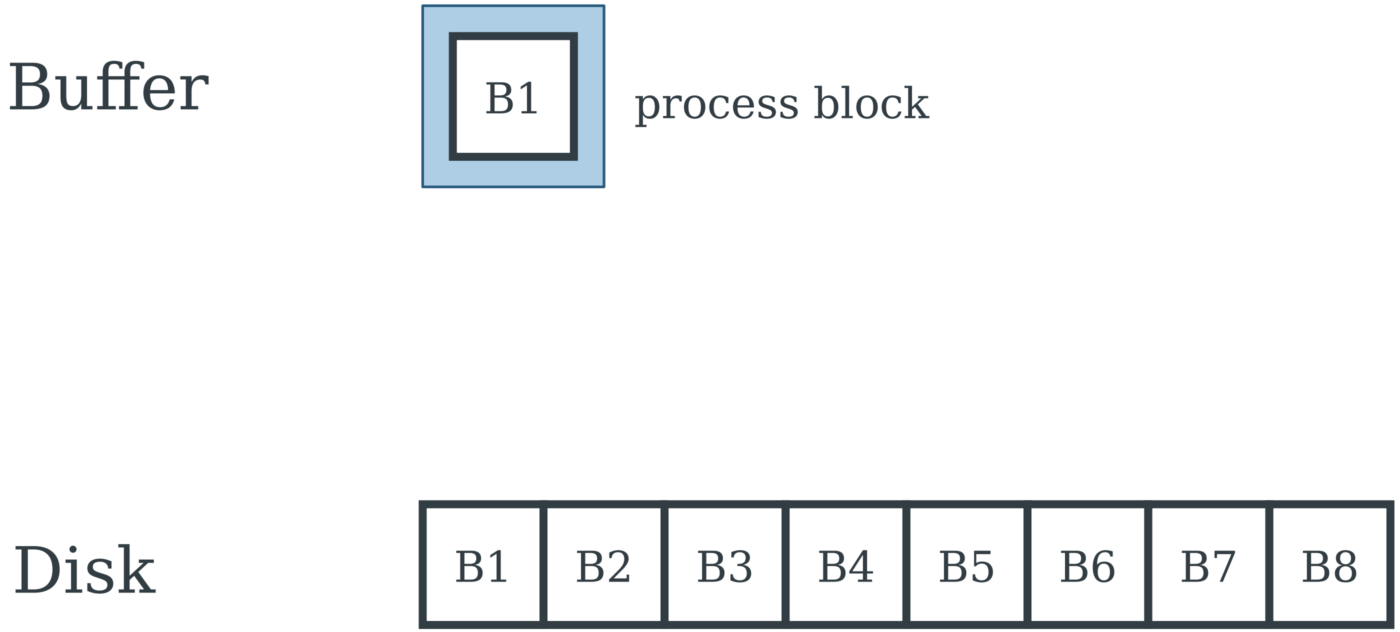
Single Buffering

1. Read B1 $\rightarrow$ Buffer
2. Process Data in Buffer
3. Read B2 $\rightarrow$ Buffer
4. Process Data in Buffer ...

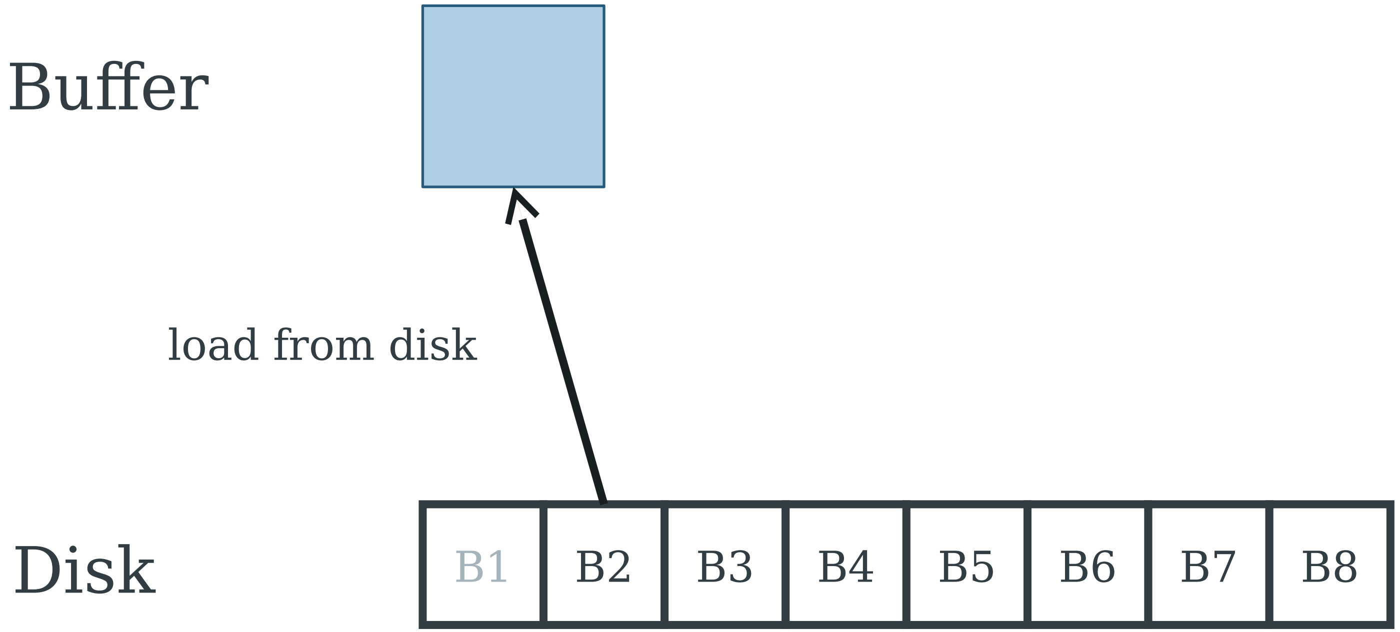
Single Buffering



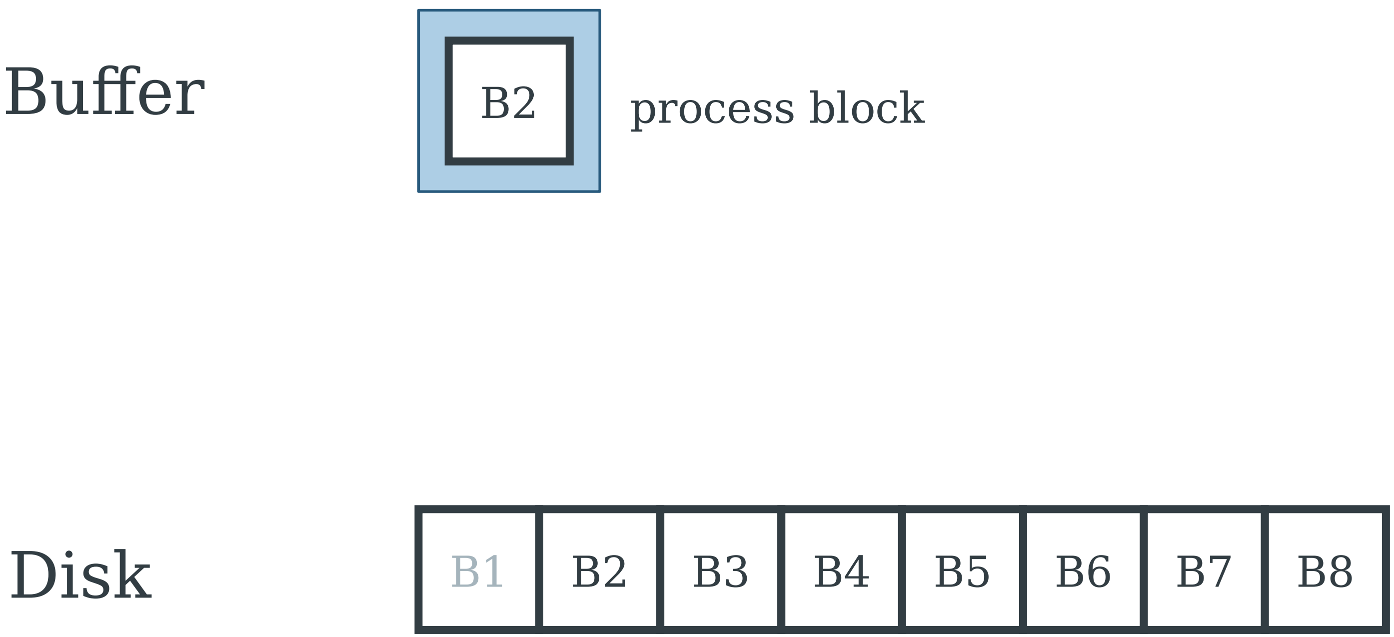
Single Buffering



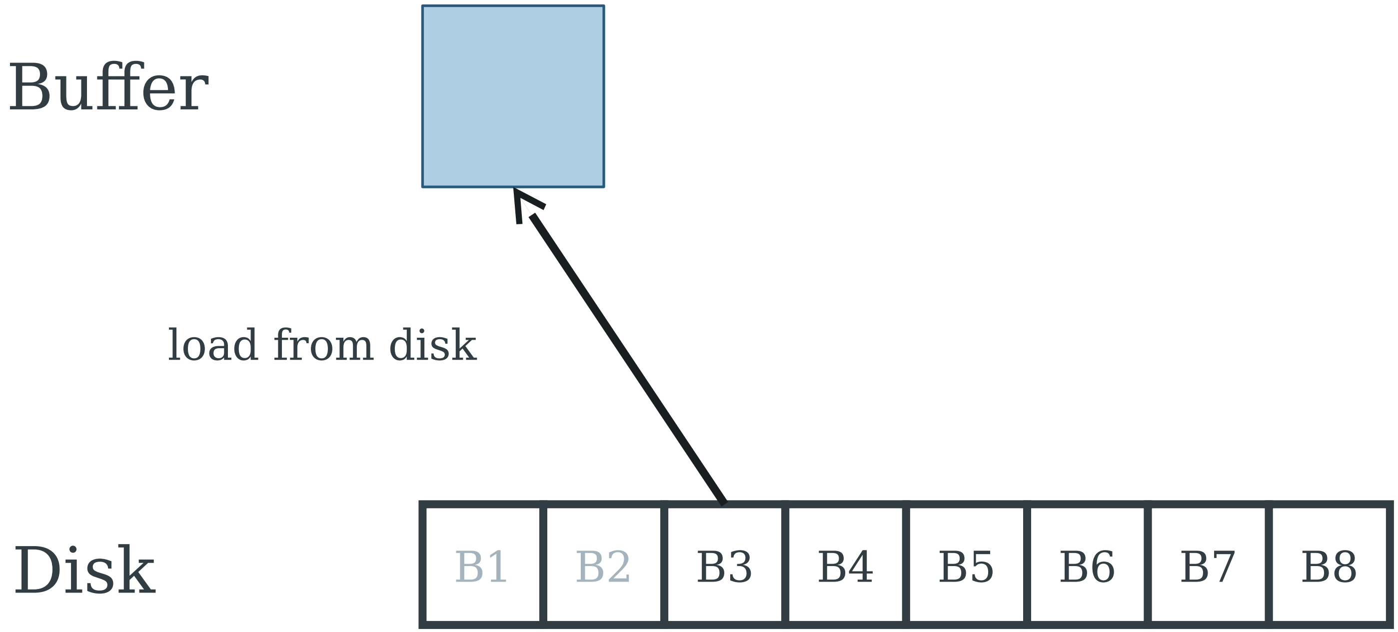
Single Buffering



Single Buffering



Single Buffering



Single Buffering Cost

Single buffer time = $n(P + R)$

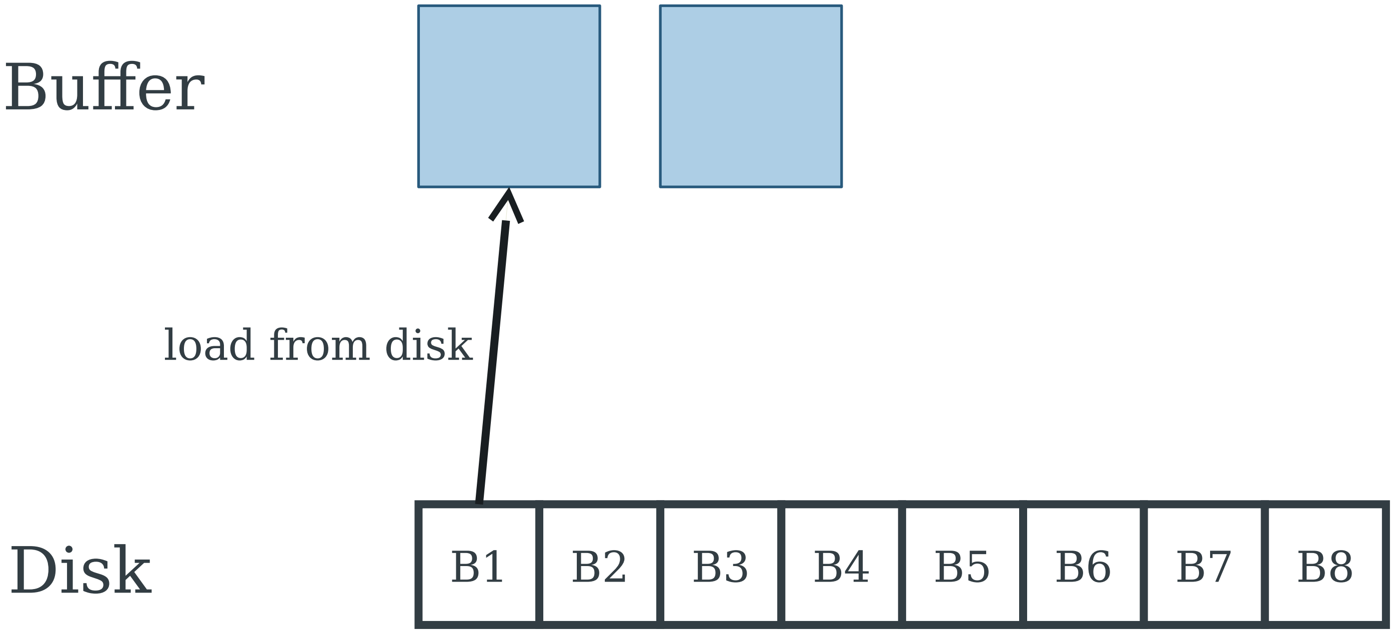
where P = time to process a block
R = time to read a block
n = number of blocks

Double Buffering

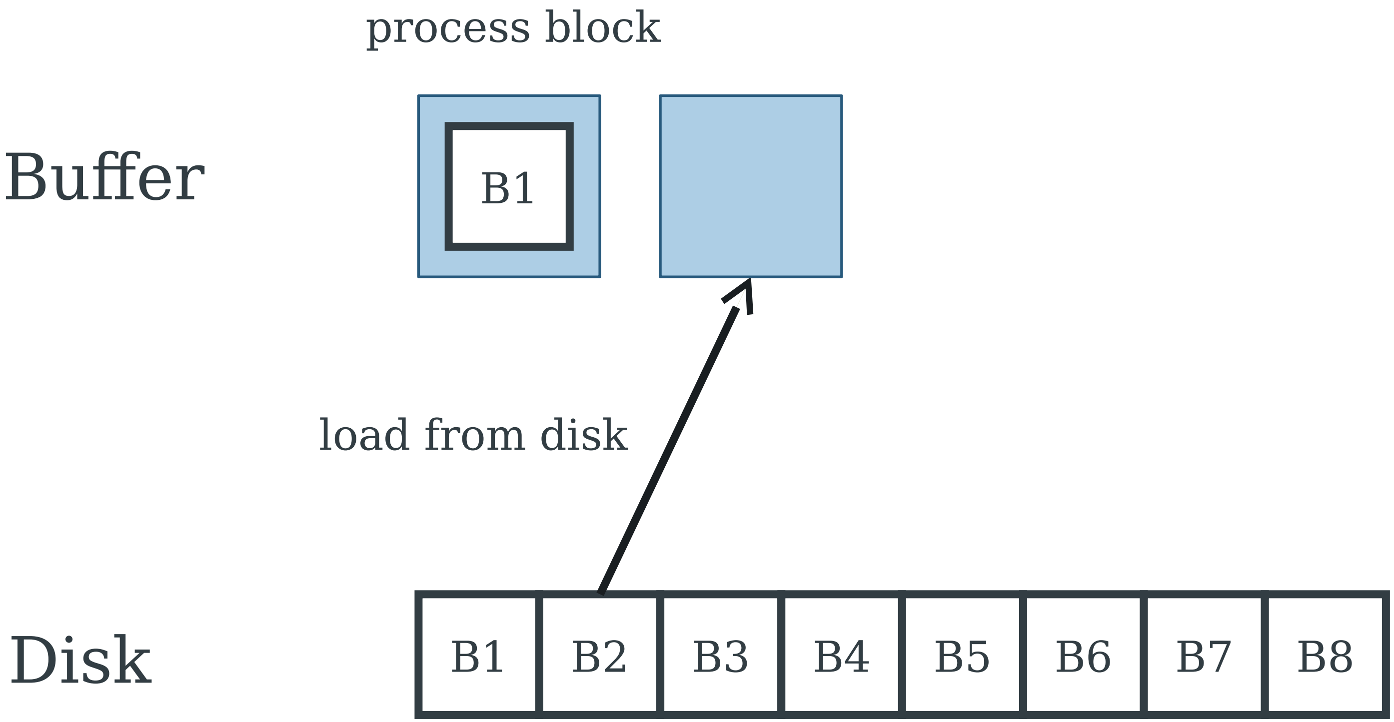
Use a pair of buffers:

- While reading a block and writing into buffer A
- Process block previously read into buffer B
- After block read into A, process A and read next block into B

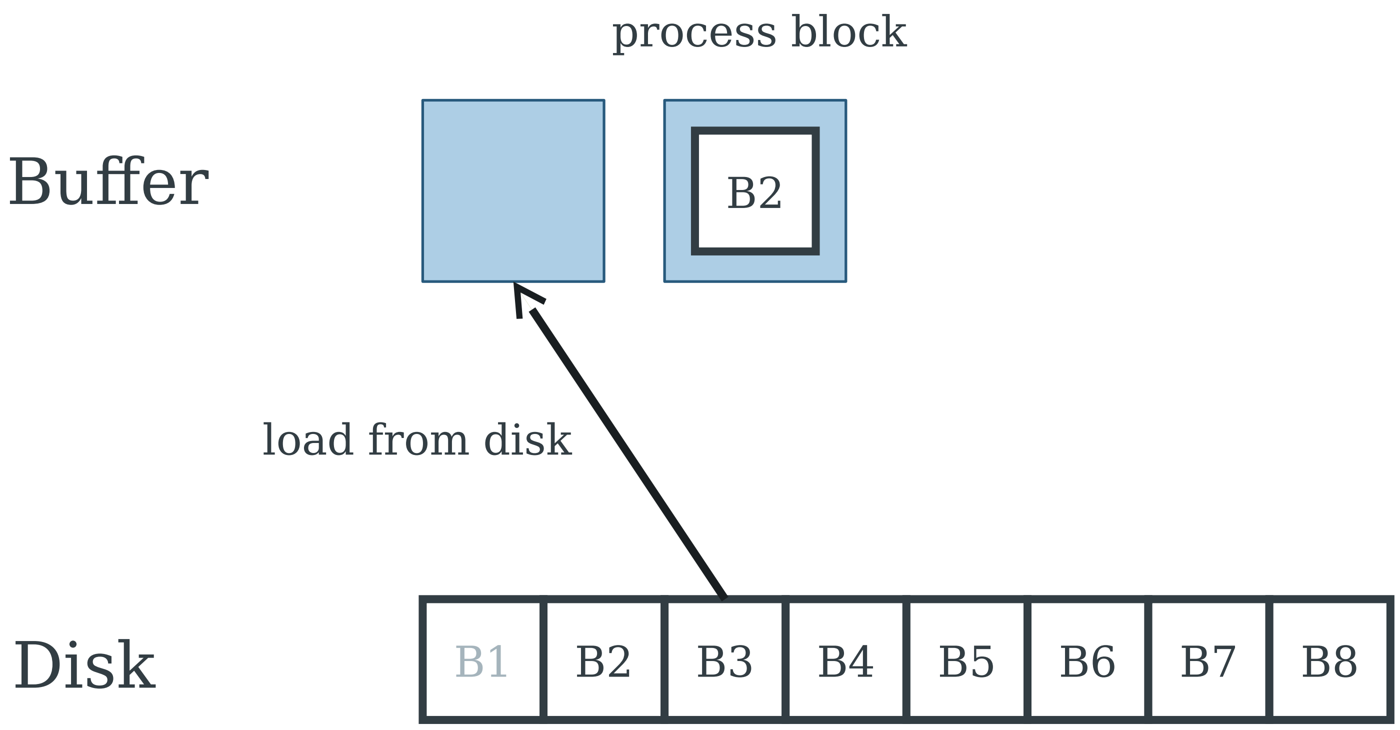
Double Buffering



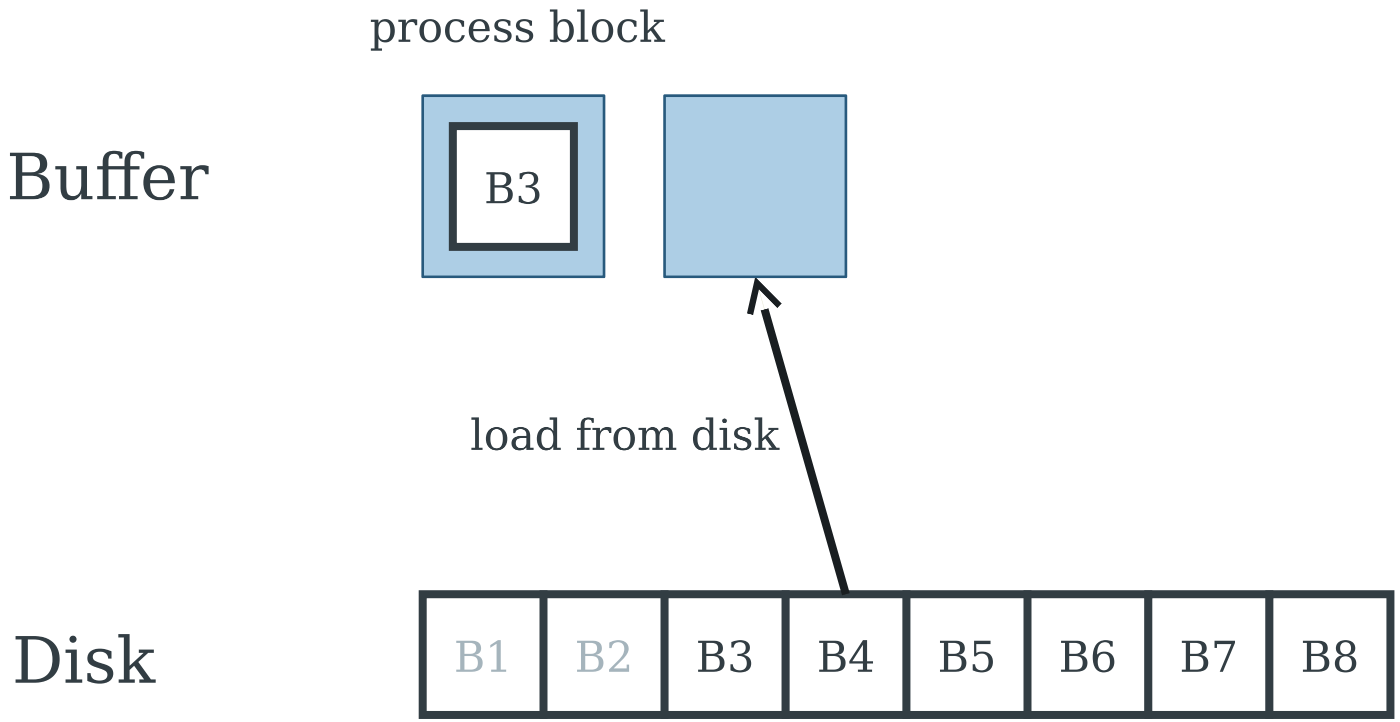
Double Buffering



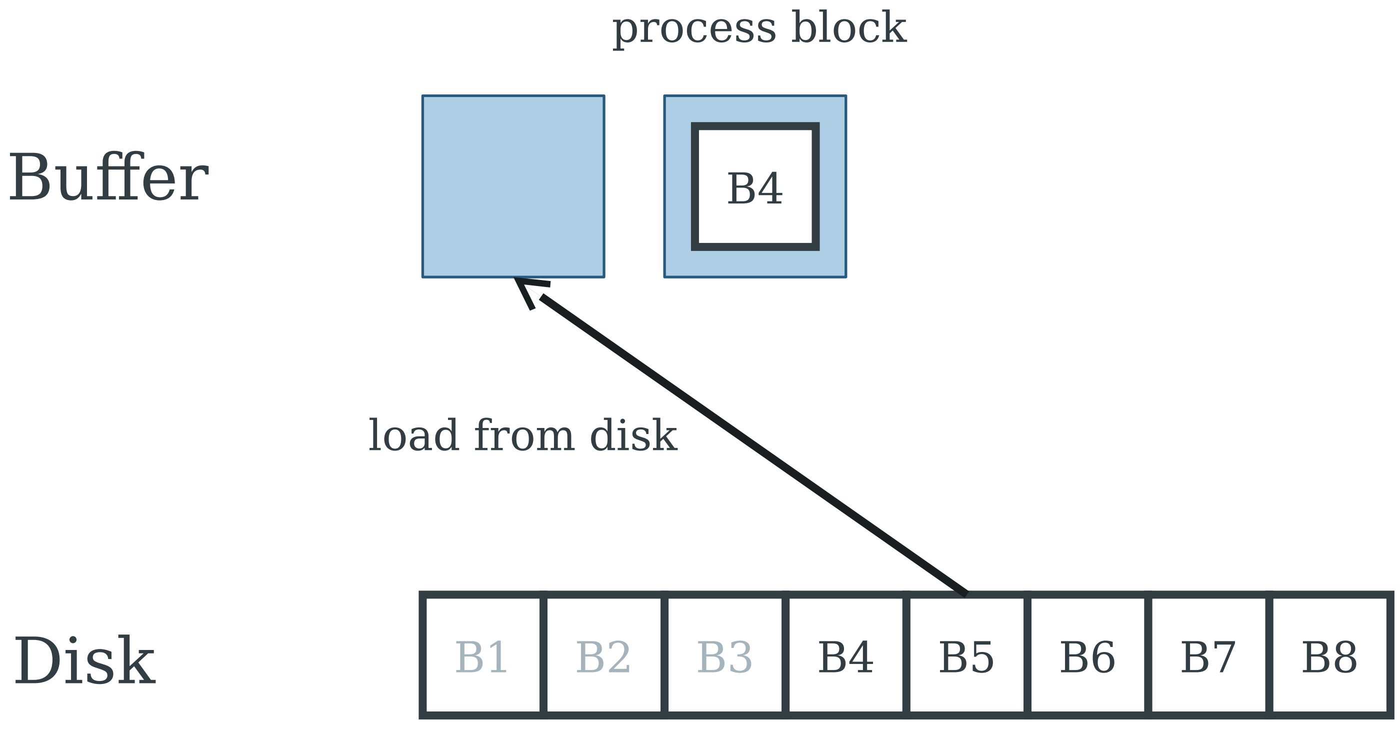
Double Buffering



Double Buffering



Double Buffering



Double Buffering

If time to process a block > time to read a block:

Double buffer time = R + nP

Single buffer time = n(R+P)

The Five Minute Rule

The Five Minute Rule

Data referenced every five minutes should be memory resident

The Five Minute Rule

The Five Minute Rule for trading memory for disc accesses
Jim Gray & Franco Putzolu
May 1985

The Five Minute Rule, Ten Years Later
Goetz Graefe & Jim Gray
December 1997

The five-minute rule 20 years later (and how flash memory changes the rules)
Goetz Graefe
July 2009

The Five Minute Rule

Assume a block is accessed every X seconds:

CD = cost if we keep that block on disk

- $D = cost of disk unit
- I = number of IOs that unit can perform per second
- In X seconds, unit can do XI IOs
- So, CD = $D / XI

The Five Minute Rule

Assume a block is accessed every X seconds:

CM = cost if we keep that block in RAM

- \$M = cost of 1MB of RAM
- P = number of pages in 1MB RAM
- So   CM = \$M / P

The Five Minute Rule

Assume a block is accessed every X seconds:

If CD is smaller than CM,

- keep block on disk
- else keep in memory

Break even point when CD = CM, or X = ($D P) / (I $M)

Using 1997 numbers

P = 128 blocks/MB (8KB pages)

I = 64 accesses/sec/disk

$D = $2000/disk (9GB HDD + controller)

$M = $15/MB of RAM

X = 266 seconds (about 5 minutes)
(did not change much from 1985 to 1997)

Using 2007 numbers

P = 256 blocks/MB (4KB pages)

I = 83 accesses/sec/disk (12ms to read 4KB)

$D = $80/disk (250GB SATA HDD)

$M = $0.047/MB of RAM

X = 5,248 seconds (about 1.5 hours)

Using 2007 numbers

P = 256 blocks/MB (4KB pages)

I = 6,200 accesses/sec/disk (0.16ms to read 4KB)

$D = $999/disk (32GB SSD)

$M = $0.047/MB of RAM

X = 876 seconds (about 15 minutes)

Using 2016 numbers

P = 256 blocks/MB (4KB pages)

I = 64,000 accesses/sec/disk (0.015ms to read 4KB)

\$D = \$685/disk (240GB SSD)

\$M = \$0.034/MB of RAM

X = 805 seconds (about 13.5 minutes)

Disk Organisation

Overview

- Data Items
- Records
- Blocks
- Files

Data Items

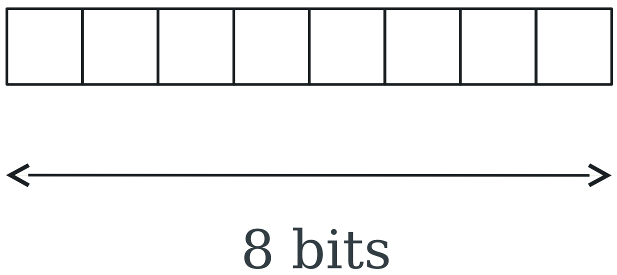
Data Items

We might wish to store:

- a salary
- a name
- a date
- a picture

Data Items

We have: bytes



Representing numbers

Integer (short): 2 bytes

- e.g. 57 is

0	0	0	0	0	0	0	0

0	0	1	1	1	0	0	1

Real numbers: IEEE 754 (floating point)

- 1 bit sign, n bits for mantissa, m bits for exponent

Representing characters

Various coding schemes: ASCII, utf-8

- 'A'	0	1	0	0	0	0	0	1
- 'c'	0	1	1	0	0	0	1	1
- CR	0	0	0	0	1	1	0	1

Representing booleans

1 byte per value

- True

1	1	1	1	1	1	1	1

- False

0	0	0	0	0	0	0	0

We can pack more than one value per byte, if we’re desperate

Representing dates

Days since a given date (integer)

- 1st Jan 1900
- 1st Jan 1970 (UNIX epoch)

ISO8601 dates

- Calendar dates: YYYYMMDD (8 characters)
- Ordinal dates: YYYYDDD (7 characters)

Representing times

Seconds since midnight (integer)

ISO8601 times

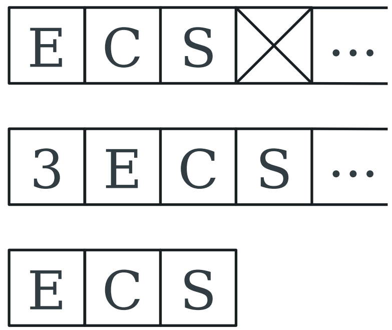
- HHMMSS (6 characters)
- HHMMSSFF (8 characters, to represent fractional seconds)

Representing strings

Null terminated

Length given

Fixed length



Representing bit arrays

length	bits

In general…

Data items are either

- Fixed length (integers, characters, etc)
- Variable length (strings, bit arrays) usually with length given at start

May also include type of data item

- Tells us how to interpret the item
- Tells us size, if fixed

Records

Records

Collection of related data items (*fields*)

e.g. Employee record consists of:

- name field
- salary field
- employment start date field

Record types

Records may have fixed or variable formats

Records may have fixed or variable lengths

Fixed format records

Schema describes the structure of records:

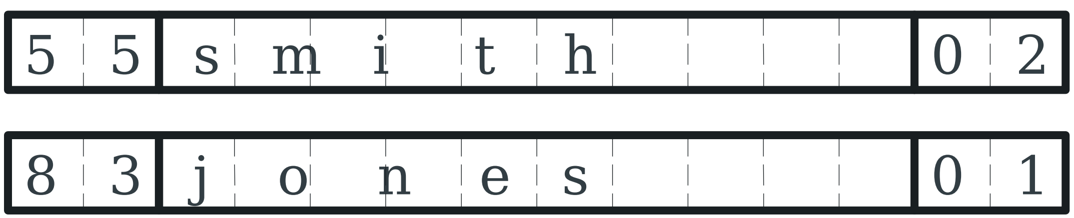
- number of fields
- types of fields
- order in record
- meaning of each field

Example: Fixed format record

Employee record structure:

1. e#, 2 byte integer
2. name, 10 char
3. dept, 2 byte code

schema



records

Variable format records

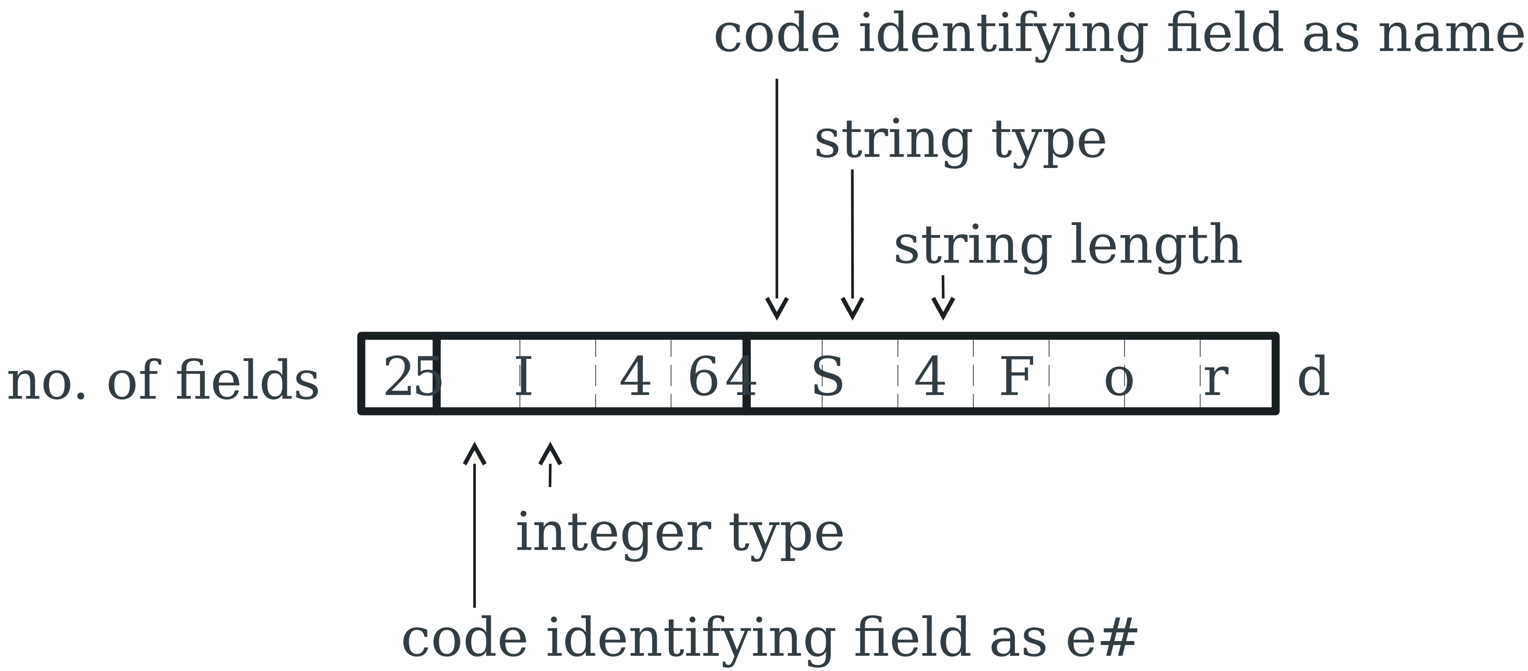
Schema-less format

- Record itself contains format: “self-describing”

Useful for sparse records, repeating fields, evolving formats

May waste space compared to a fixed format records

Example: Variable format record



Record headers

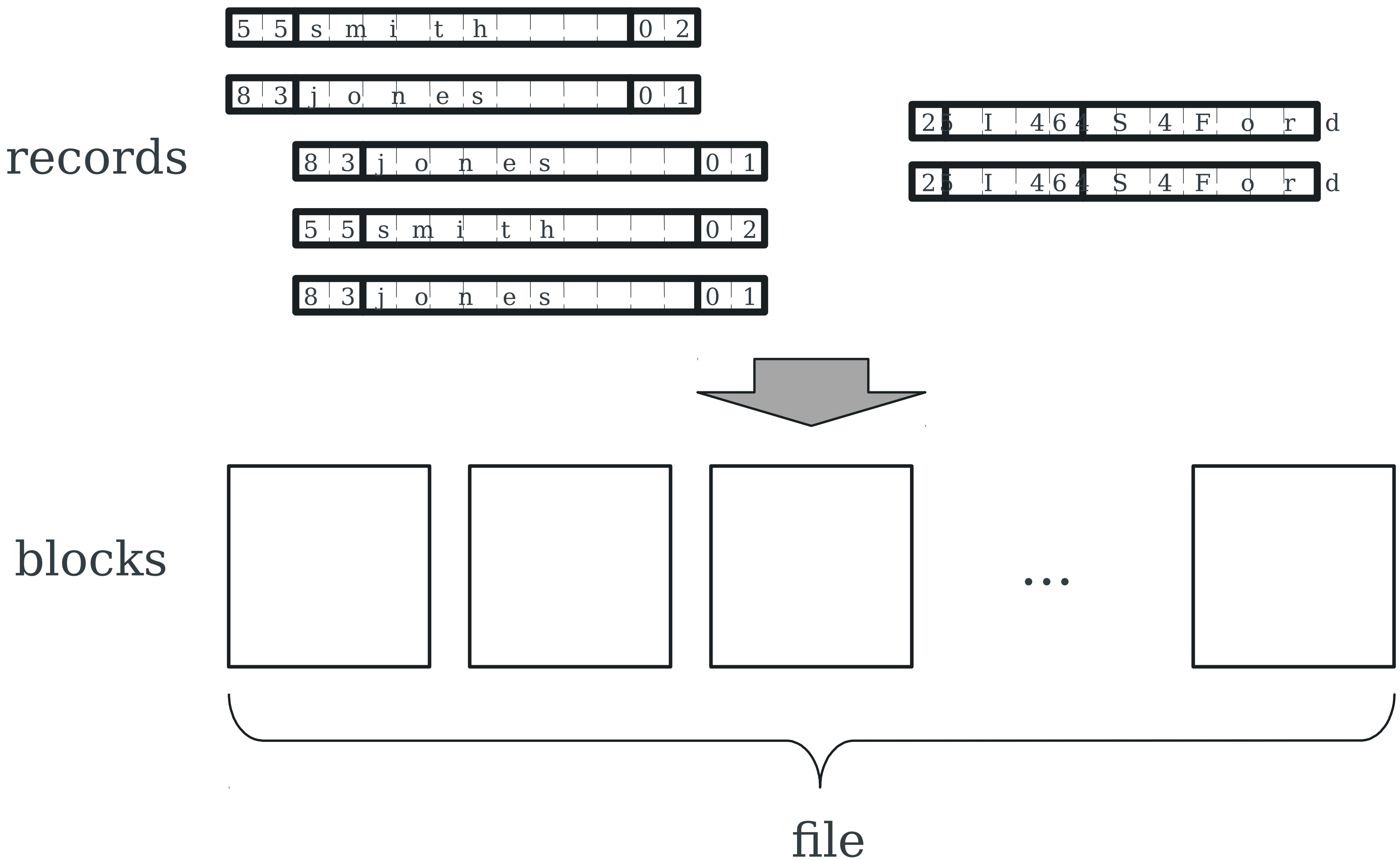
Data at beginning of record that describes record:

- record type (points to schema)
- record length
- timestamp

Intermediate between fixed and variable format

Blocks

Storing records in blocks



Block header

Data at beginning that describes block

May contain:

- File ID (or RELATION or DB ID)
- This block ID
- Record directory
- Pointer to free space
- Type of block (e.g. contains recs type 4; is overflow, …)
- Pointer to other blocks “like it”
- Timestamp …

Placing records in blocks

Considerations:

- separating records
- spanned vs. unspanned
- sequencing
- indirection

Separating records in a block

Block

R1	R2	R3

Three approaches:

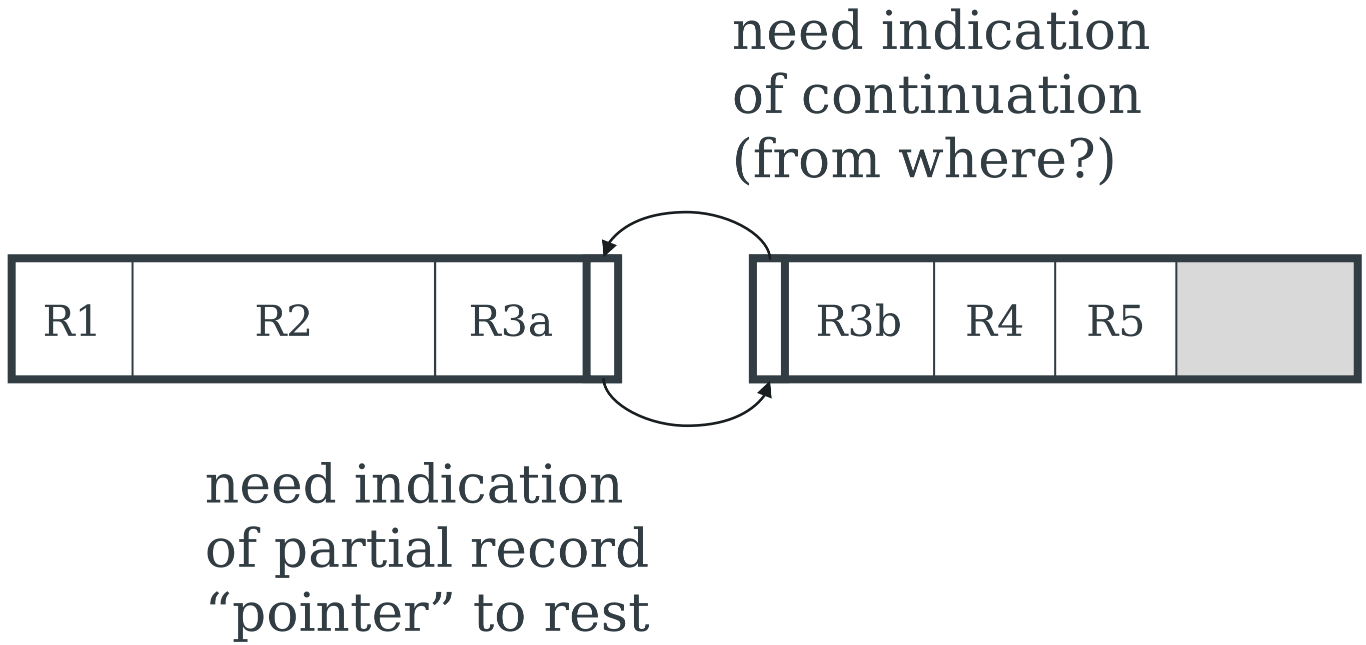
1. use fixed length records - no need to separate
2. use a special marker to indicate record end
3. give record lengths (or offsets)
   - within each record
   - in block header

Spanned vs. Unspanned

Unspanned: each record must fit within a single block

Spanned: records may be split between blocks

Spanned records



Spanned vs. Unspanned

Unspanned records are much simpler, but may waste space…

Spanned records are essential if record size > block size

Sequencing

Sequencing: ordering records in file (and block) by some key value

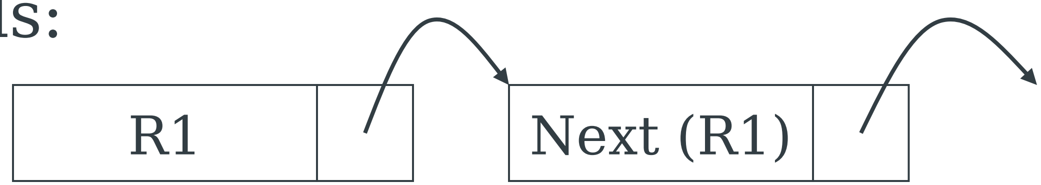
Makes it possible to efficiently read records in order

- e.g., to do a merge-join — discussed later in module

Sequencing Options

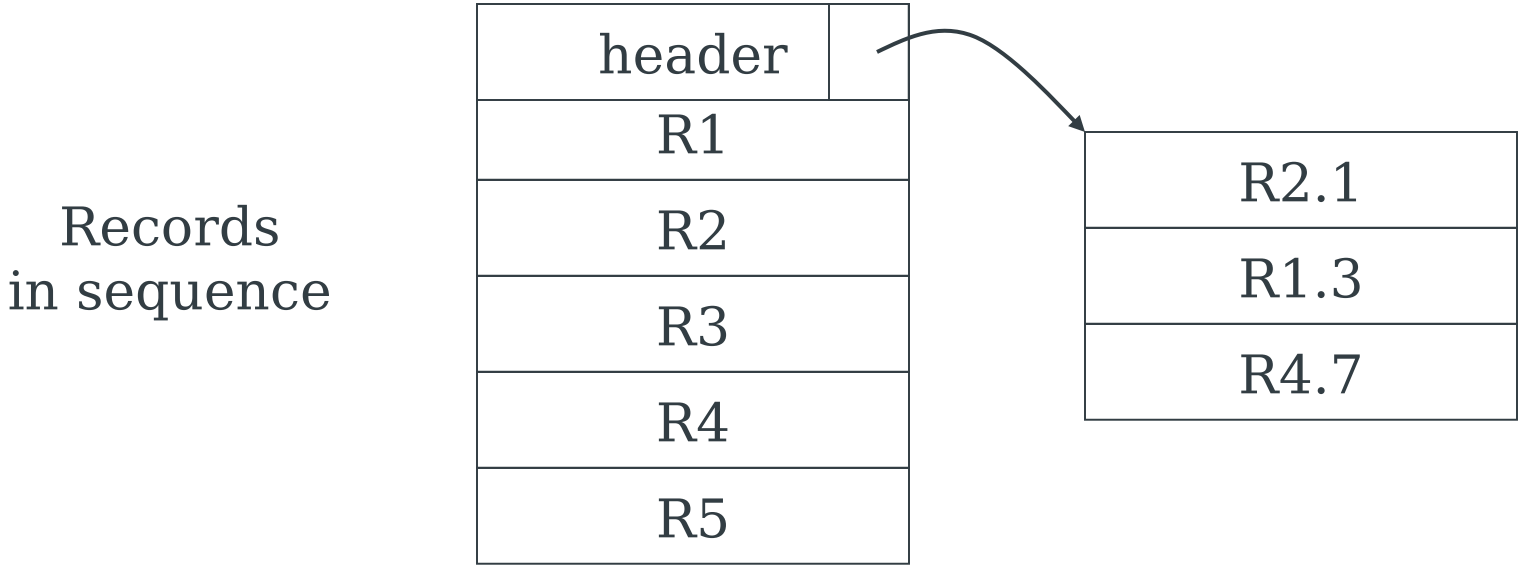
Next record physically contiguous:

Linked records:



Sequencing Options

Overflow area



Indirection

How do we refer to records?

Many options:

- physical addressing
- indirect addressing
- other options in between

Tradeoff between:

- flexibility (easier to move records on insertion/deletion)
- cost (of maintaining indirection)

Physical Addressing

Record ID =

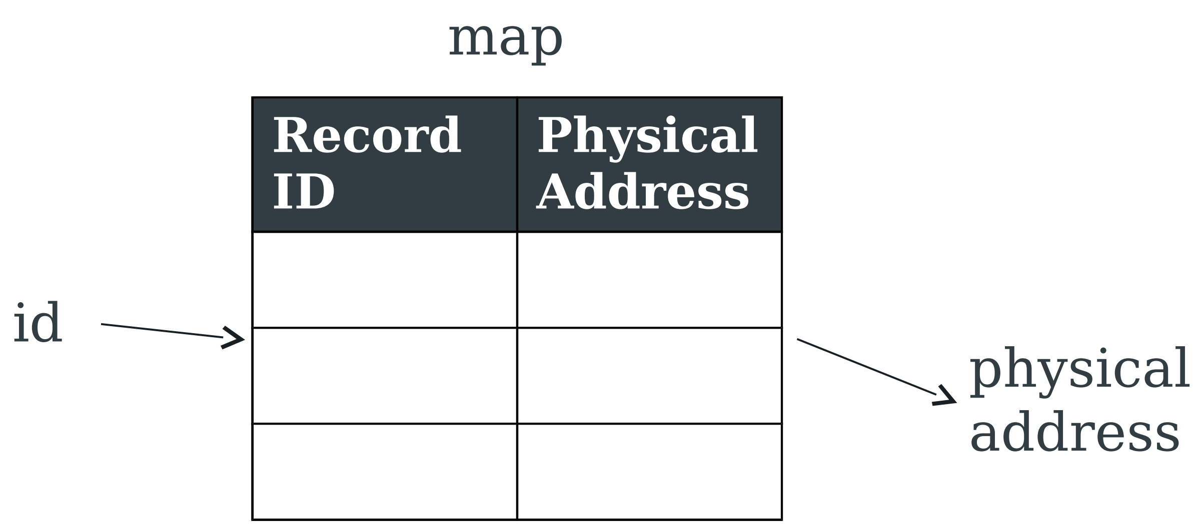
Device
Cylinder
Head
Sector

Block ID

Offset in block

Indirect Addressing

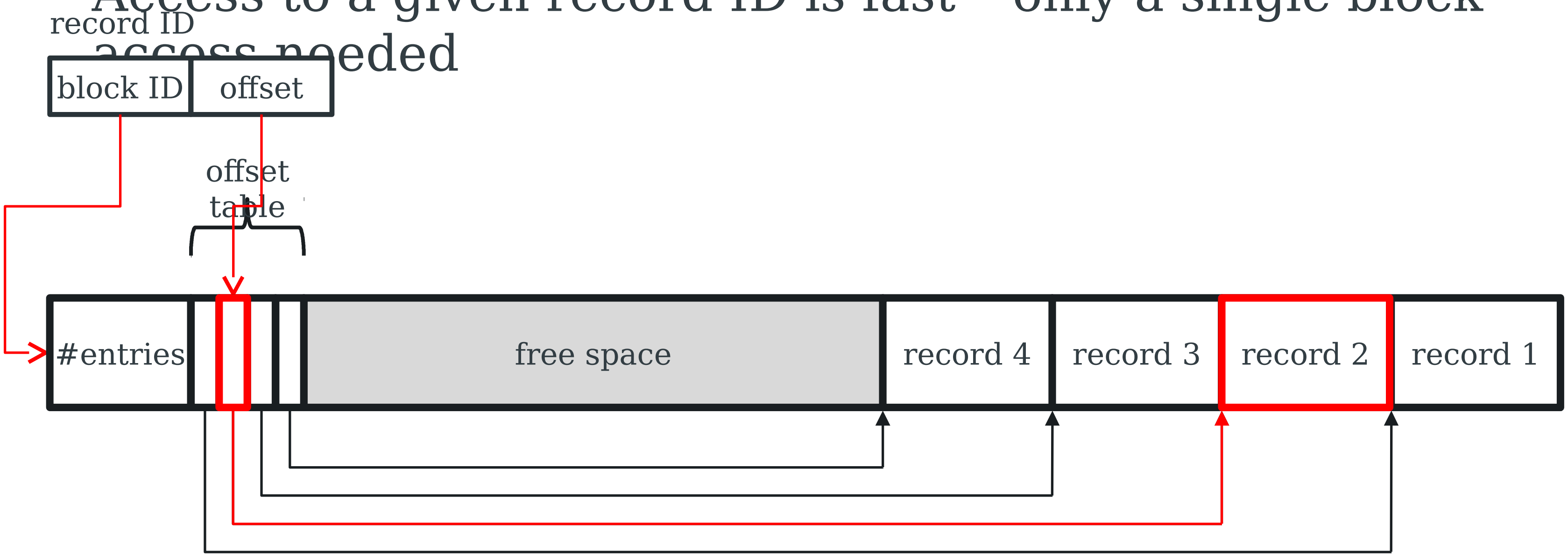
Record ID is arbitrary bit string



Indirection in block

Typical implementation

- Records can be shifted within block without changing record ID
- Access to a given record ID is fast – only a single block access needed



Address Management

Every block and record has *two* addresses:

- a database address (when in secondary storage)
- a memory address (when copied into a buffer)

When in a buffer, using memory addresses (= pointers) is more efficient

Otherwise, translation table is required:

- converts database address
- into current memory address

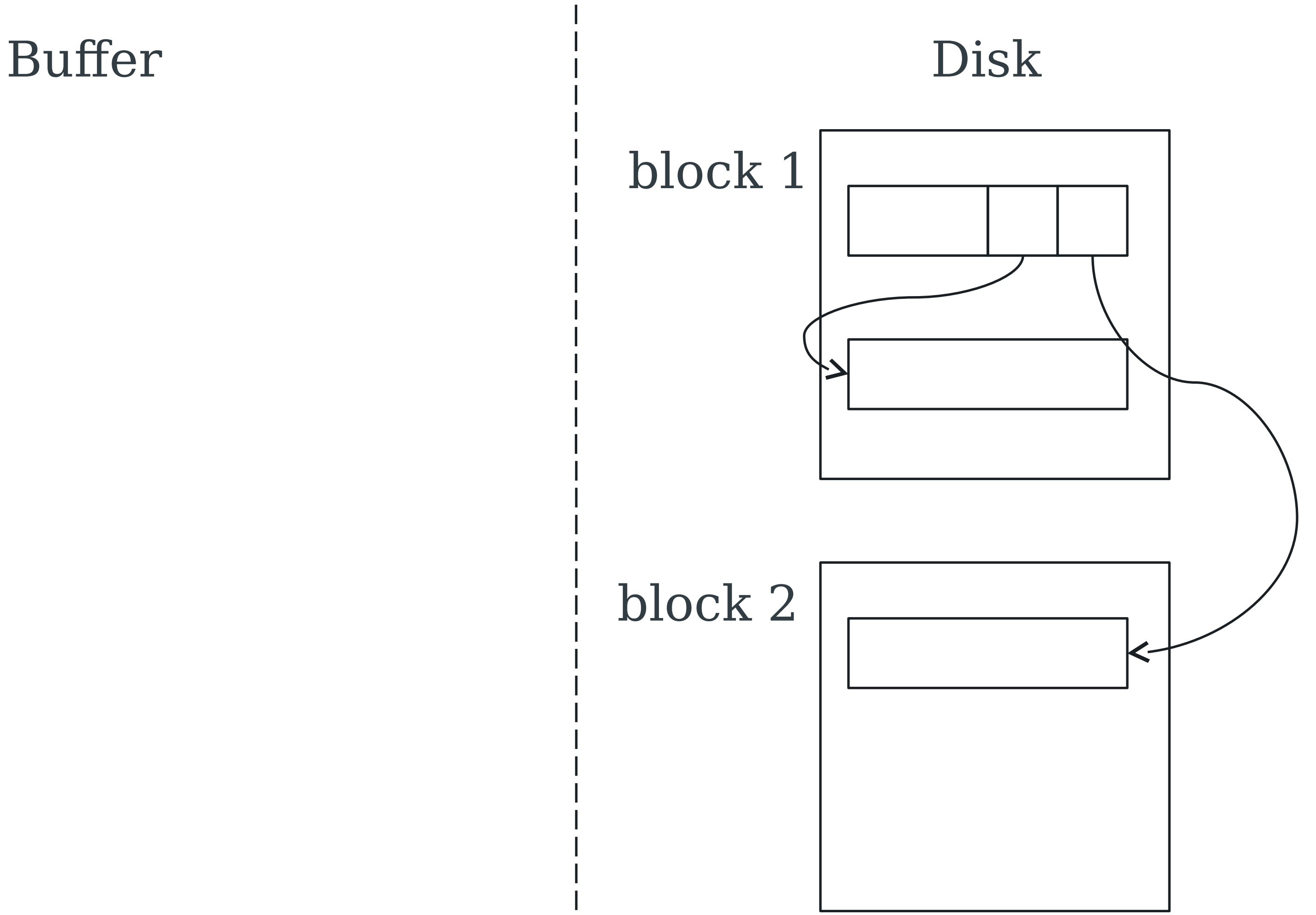
Pointer Swizzling

General term for techniques used to translate database address space to virtual memory address space

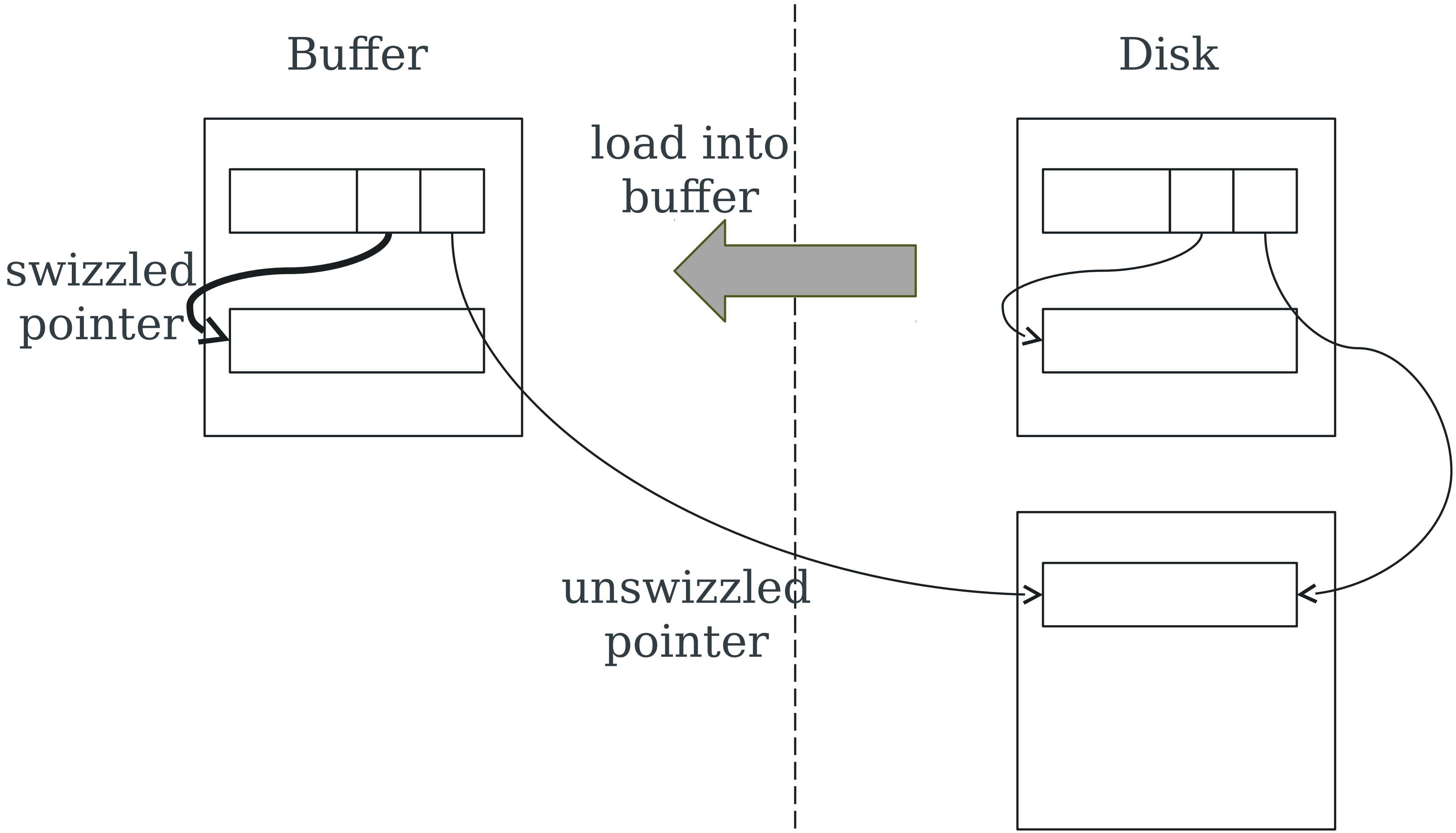
Swizzled pointers consist of

- One bit to indicate whether the pointer is a database address or a memory address
- Database or memory pointer, as appropriate

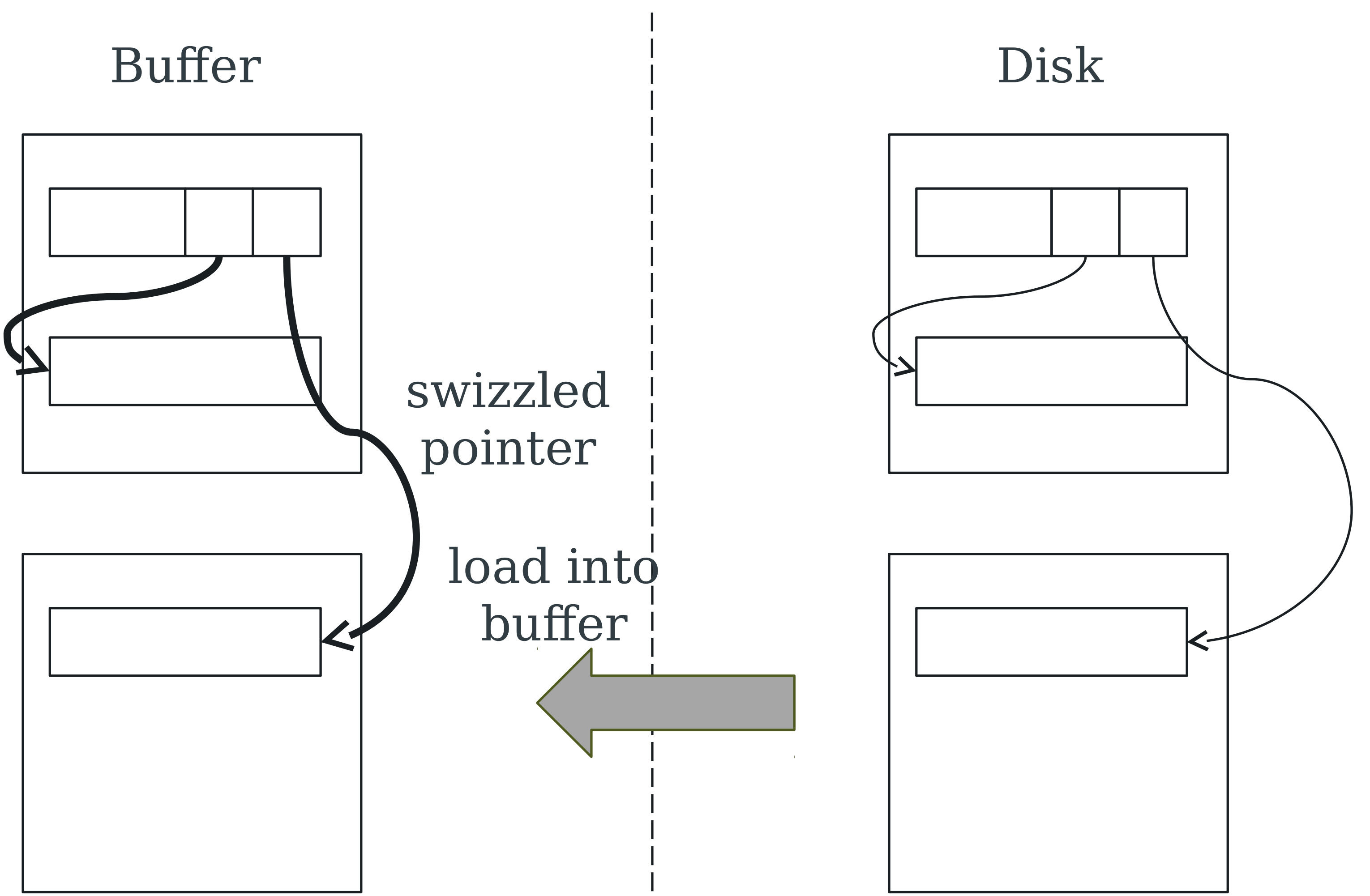
Swizzling



Swizzling


Buffer
Disk
load into buffer
swizzled pointer
unswizzled pointer

Swizzling



Swizzling Strategies

Automatic

- As soon as block brought into memory, locate all pointers and addresses and enter them into translation table
- Replace pointers in blocks with new entries

On Demand

- Leave all pointers unswizzled when block in brought into memory
- Swizzle pointers only when dereferenced

No swizzling

- Use translation table to map pointers on each dereference

Unswizzling

Reverse of the swizzling operation

- When a block is written back to disk, rewrite swizzled pointers using the translation table
- Need to beware of pinned blocks (that cannot yet be safely written to disk)

Insertion and Deletion

Insertion: the easy case

Records not in sequence

- Insert new record at end of file or in deleted slot
- If records are variable size, not as easy...

Insertion: the hard case

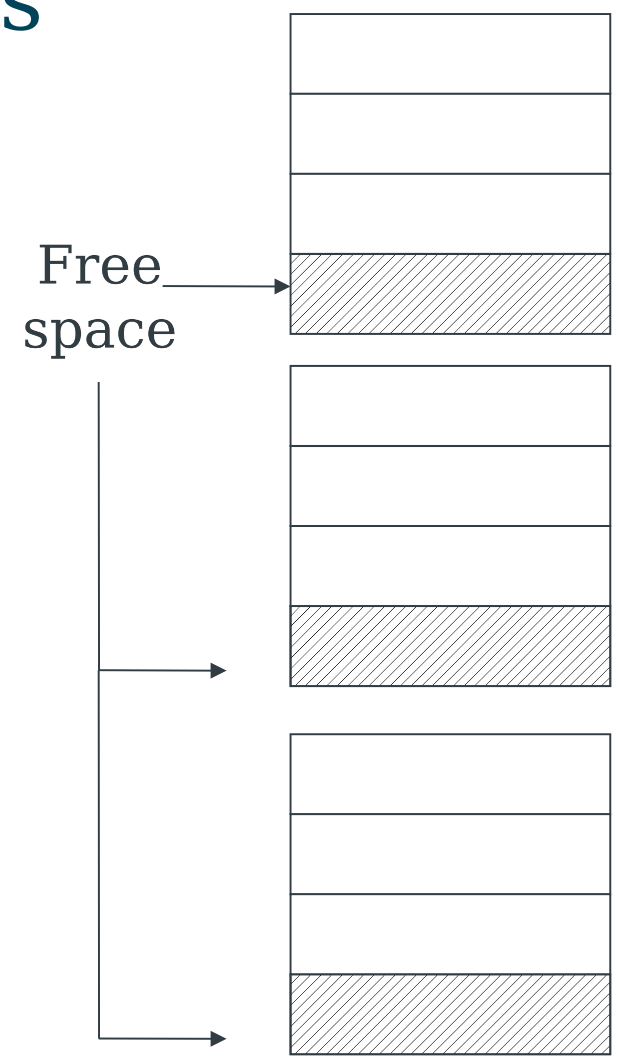
Records in sequence

- If free space “close by”, not too bad…
- Or use overflow idea…

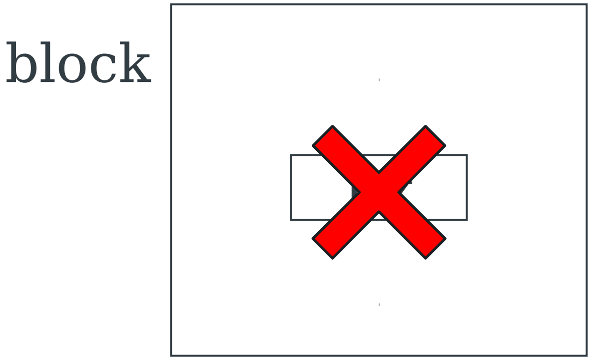
Insertion considerations

How much free space to leave in each block, track, cylinder?

How often should we reorganize file + overflow?



Deletion



Two main options:

- Immediately reclaim space
- Mark space as deleted

Deletion marking

May need a chain of deleted records (for re-use)

Need a way to mark deleted records:

- special characters
- delete field
- in map

Deletion tradeoffs

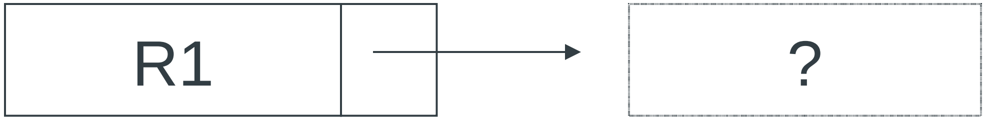
How expensive is it to move valid record to free space for immediate reclaim?

How much space is wasted?

- e.g., deleted records, delete fields, free space chains,…

Deletion considerations

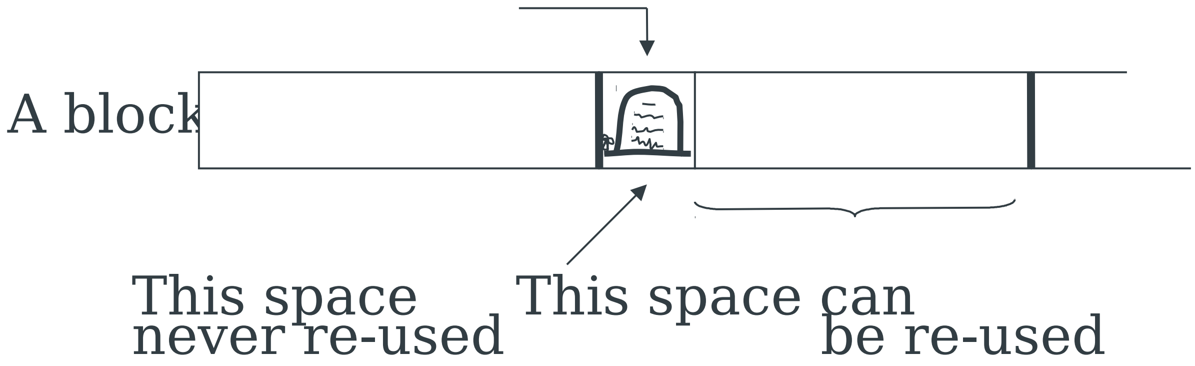
- How do we deal with dangling pointers?



Tombstones

Leave “MARK” in map or old location

Physical IDs

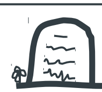


Tombstones

Leave “MARK” in map or old location

Logical IDs

map

ID	LOC
7788	

Never reuse
ID 7788 nor
space in map…

Further Reading

Further Reading

- Chapter 13 of Garcia-Molina et al
- Gray, J. and Putzolu, F. 1987. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for CPU time. Proceedings of SIGMOD 1987, 395-398.
- Gray, J. and Graefe, G. 1997. The five-minute rule ten years later, and other computer storage rules of thumb. *SIGMOD Record*. 26(4), 63-68.
- Graefe, G. 2009. The five-minute rule 20 years later (and how flash memory changes the rules). *Communications of the ACM*. 52(7), 48-59.

Next Lecture: Access Structures