# Advanced Database Systems

Spring 2024

Lecture #29:

# Revision I

# Administrivia

New quiz deadline: **Thursday, 11 April at noon**

Last tutorial is this week

Final exam

- Topics covered in the lectures and tutorials, excluding guest lecture from week 10
- 6-8 questions, all mandatory
- Can use a calculator

# Plan for Today

Files, Pages, Records

Buffer Management

Sorting

Joins

# Files, Pages, Records

Tables stored as **logical files** consisting of **pages,** each containing a collection of **records**

File (corresponds to a table)

- Page (many per file)
  - Record (many per page)

The unit of access to physical disk is the page

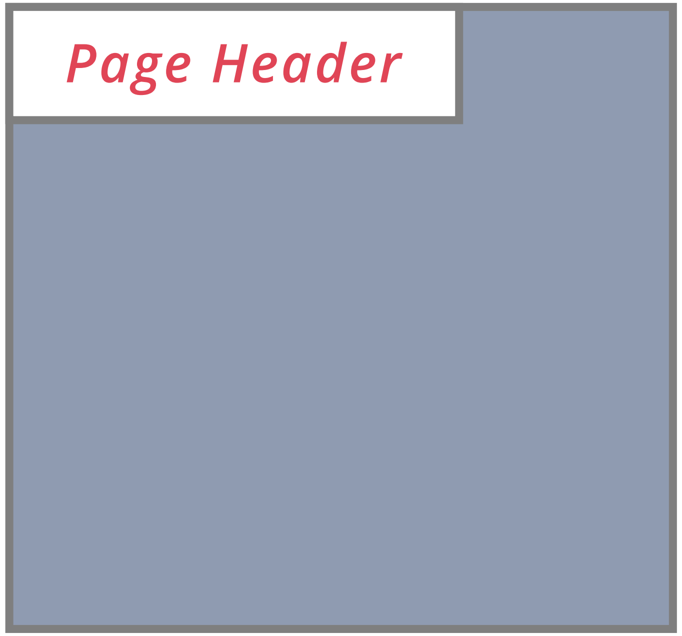
- 1 I/O = read or write 1 page

# PAGE BASICS

The page header keeps track of the records in the page

The page header may contain fields such as:

- Number of records in the page
- Pointer to segment of free space in the page
- Bitmap indicating which parts of the page are in use

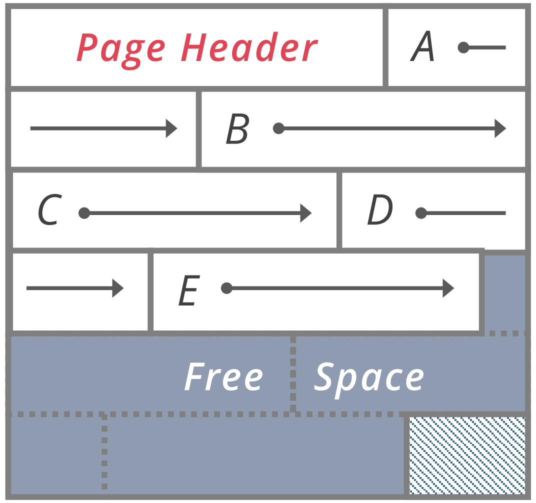



Page

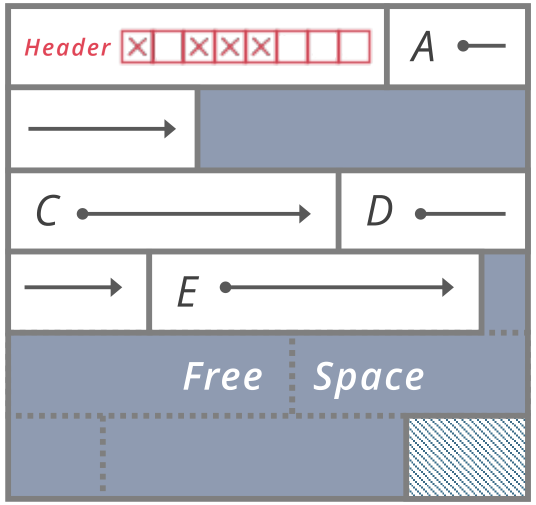
# Fixed-Length Records

**Fixed-length records** = record lengths are fixed and field lengths are consistent

**Packed Records:** no gaps between records, record ID is location in page



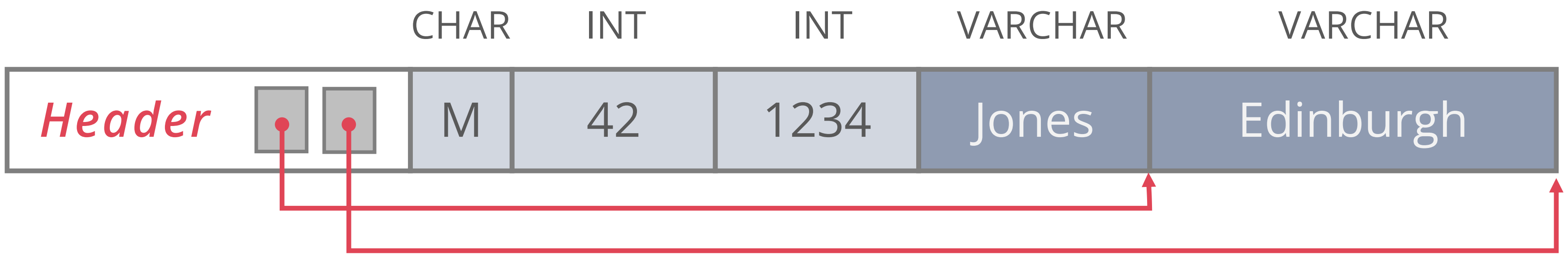

**Unpacked Records:** allow gaps between records, use a bitmap to keep track of where the gaps are



# Variable-Length Records

Variable-length records may not have fixed & consistent field lengths

We can store variable-length records with an array of field offsets:




Each record contains a **record header**

Variable length fields are placed *after* fixed length fields

Record header stores **field offset** (where variable length field ends)

# Question 1

Consider the following relation:

Assume record header stores only pointers (4B) to variable-length fields

```
CREATE TABLE Customer (
  customer_id INTEGER PRIMARY KEY,
  age INTEGER NOT NULL,
  name VARCHAR(10) NOT NULL,
  address VARCHAR(20) NOT NULL
)
```

Record header size = ???

Min record size = ???

Max record size = ???

# Question 1, Part 2

Consider the following relation:

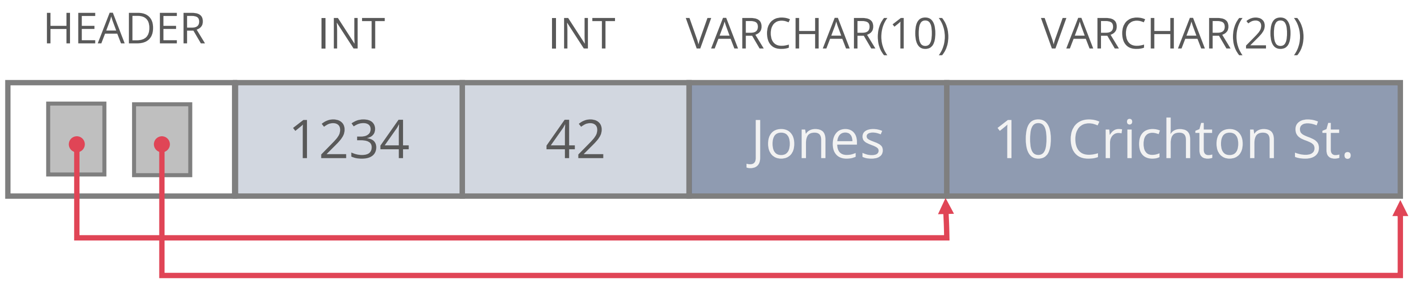
Assume record header stores only pointers (4B) to variable-length fields

```
CREATE TABLE Customer (
  customer_id INTEGER PRIMARY KEY,
  age INTEGER NOT NULL,
  name VARCHAR(10) NOT NULL,
  address VARCHAR(20) NOT NULL
)
```

Record header size = 8

Min record size = 16

Max record size = 46



# SLOTTED PAGES

Most common layout scheme is called **slotted pages**

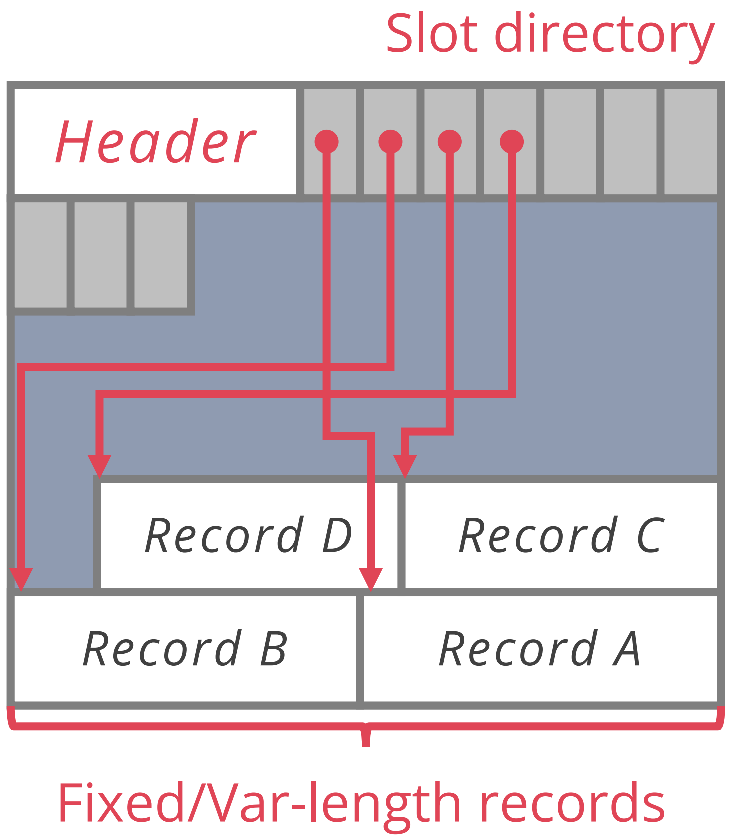
Slot directory maps "slots" to the records' starting position offsets

- Record ID = (page ID, slot ID)

Header keeps track of:

- The number of used slots
- The offset of the last slot used

Records stored at the end of page

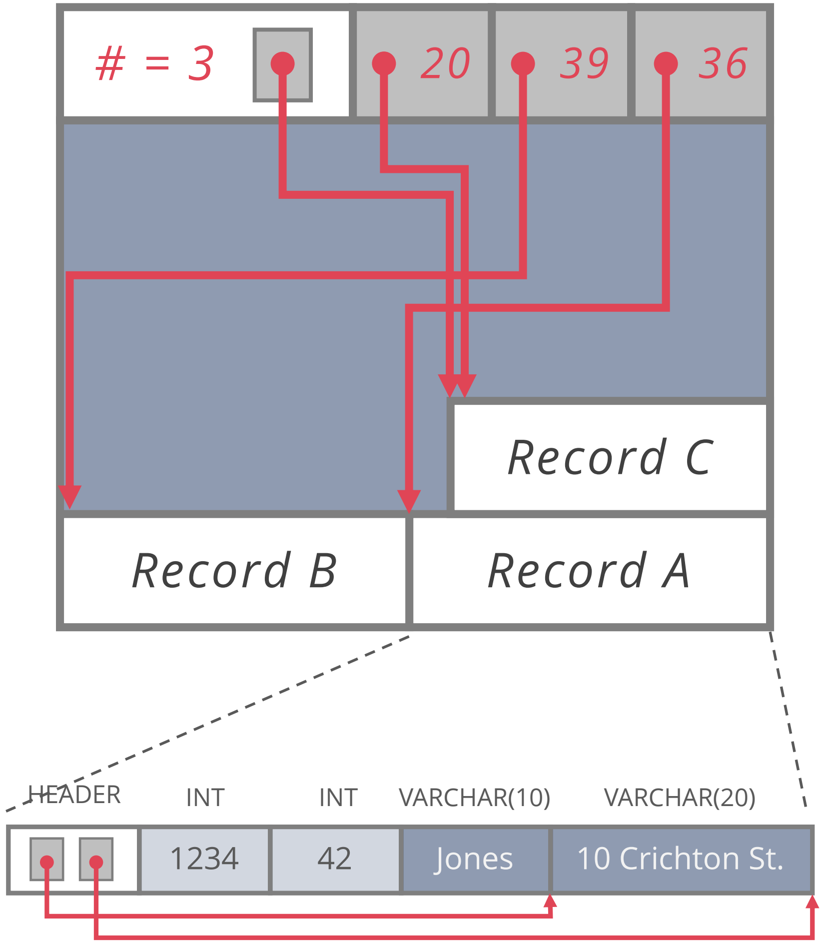


# Question 2

Suppose the Customer relation is stored using a slotted page layout

- **Page header** stores the number of records and a pointer to free space
- **Directory slot** stores a pointer and length
- **Page size** is 8KB

Max number of records = ???



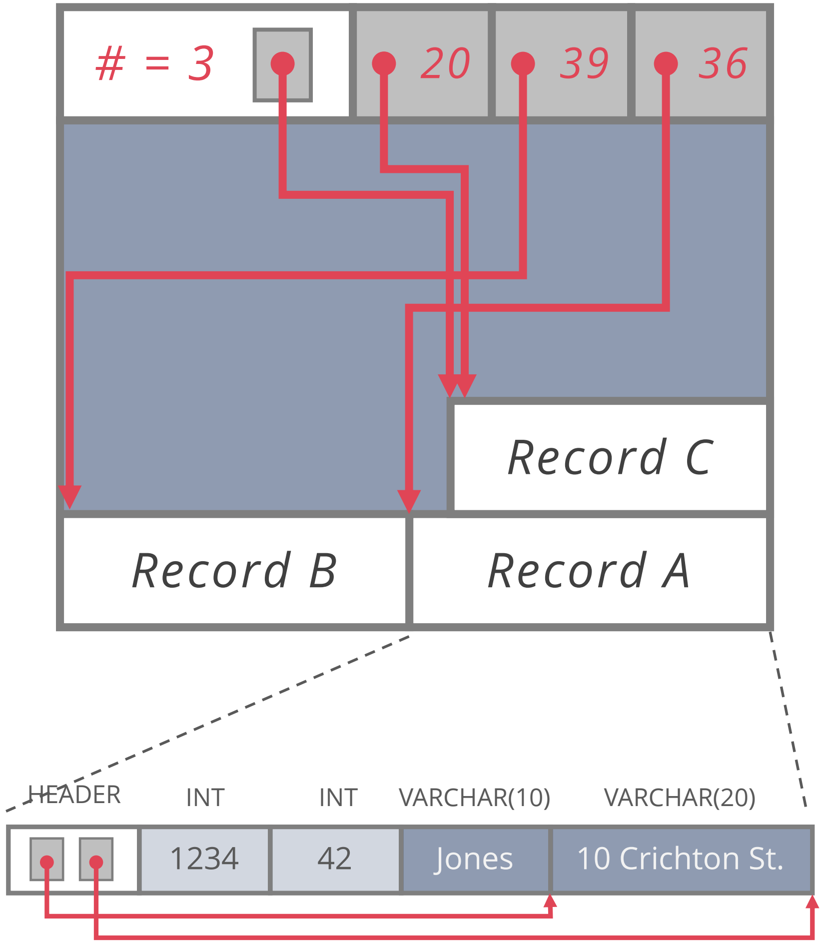
# Question 2, Part 2

Suppose the Customer relation is stored using a slotted page layout

**Page header** stores the number of records and a pointer to free space **(4B + 4B)**

**Directory slot** stores a pointer and length **(4B + 4B)**

**Page size** is 8KB




Max number of records

= (page size – header size) / (min record size + slot size)

= (8192 – 8) / (16 + 8) = **341 records**

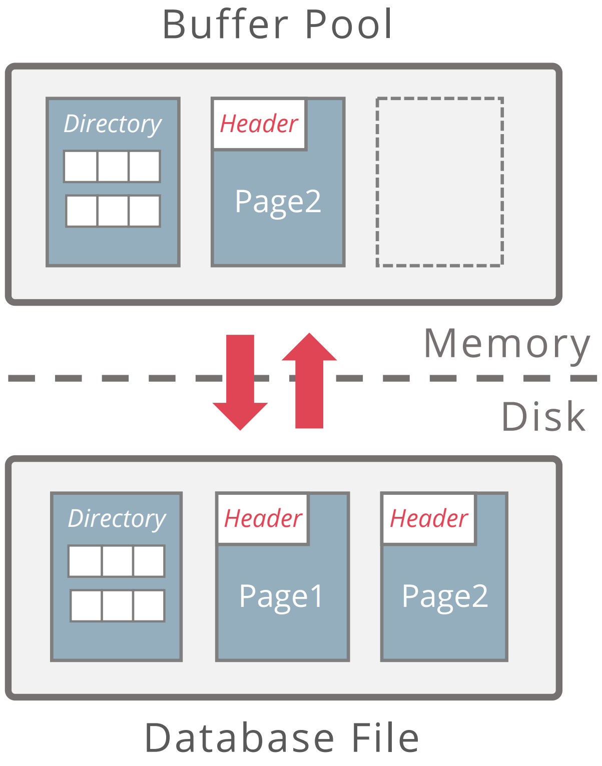
# Buffer Management

# Buffer Manager

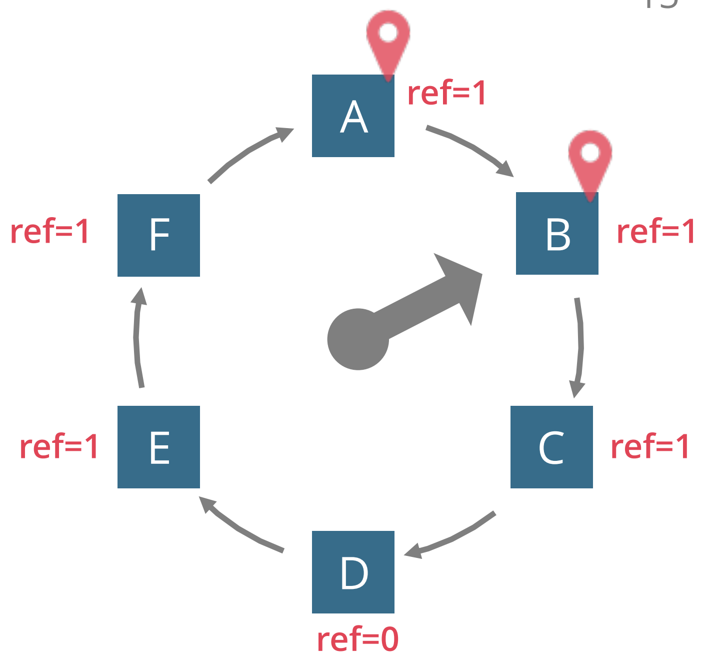
Layer that manages which pages are loaded in memory

Controls when pages are read from & written to disk

- When no space in memory, decides what page to **evict**
- Decision process is the **page replacement policy**
  - Big impact on I/Os depending on **access pattern**
- Common policies:
  - **LRU** (Least Recently Used)
  - **MRU** (Most Recently Used)
  - **Clock**



# Clock




Efficient approximation of LRU

Arrange frames in a circle (like numbers on a clock)

Advance **clock hand** around the clock to find pages to evict

- Only do this if you need to evict a page

To make this approximate least recently *used* (rather than least recently *loaded*): add a **reference bit** to each frame

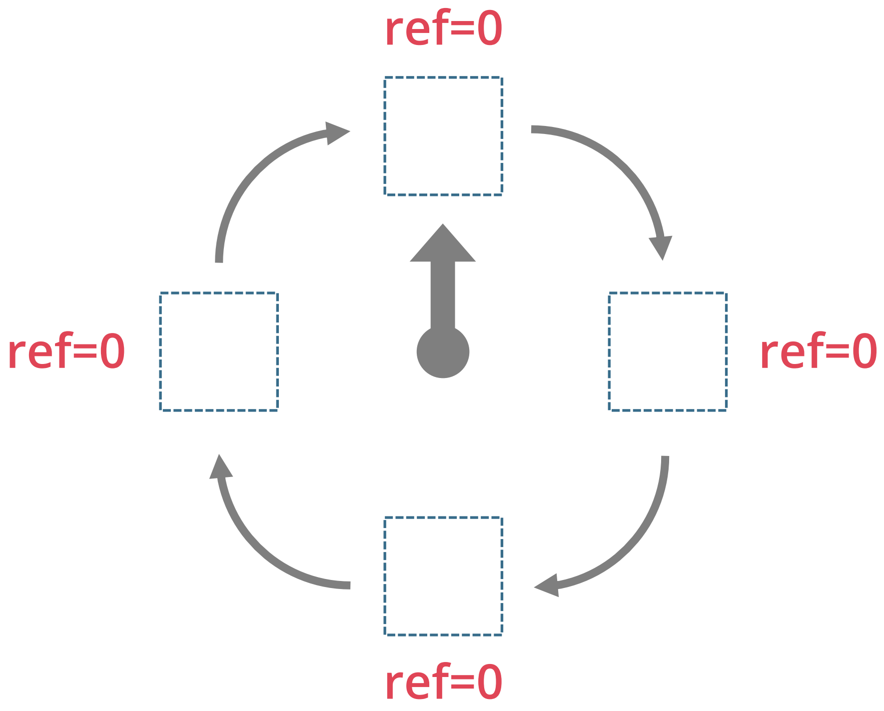
- Set to 1 on load/hit, 0 if clock hand passes the frame and the frame is unpinned
- Evict unpinned frame if clock hand reaches it and bit = 0
- (bit = 0 means less recently used than those with bit = 1)

# QUESTION 3

Page access sequence:

A B C D E B A D C A E C

Assume pages are immediately unpinned after being pinned




Buffer hits = ???

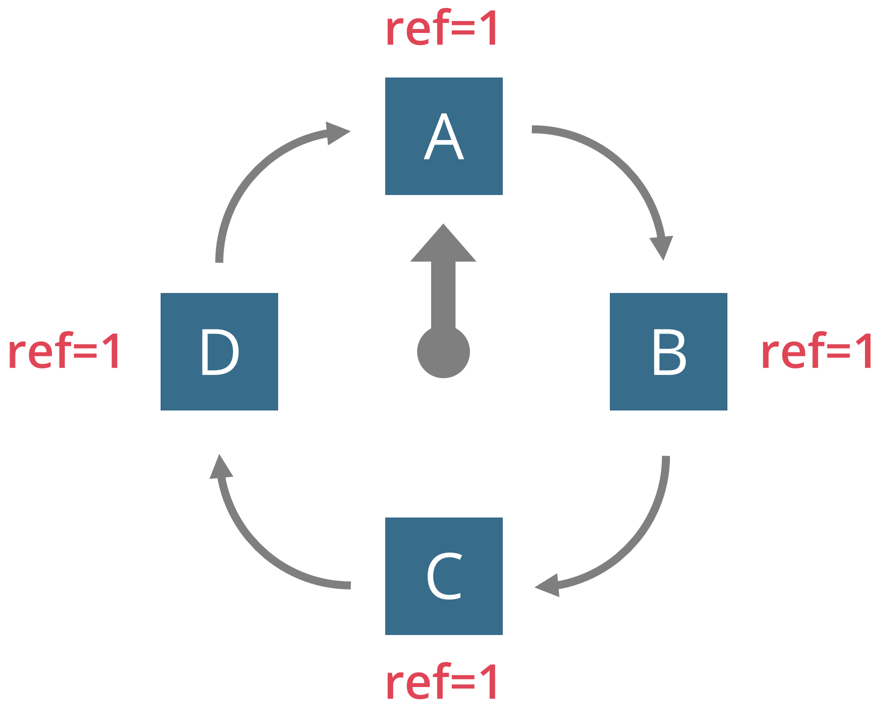
# Question 3, Part 2

Page access sequence:

A B C D E B A D C A E C

Pages A, B, C, D populate the buffer pool

The clock hand stays still




Buffer hits (so far) = 0

# Question 3, Part 3

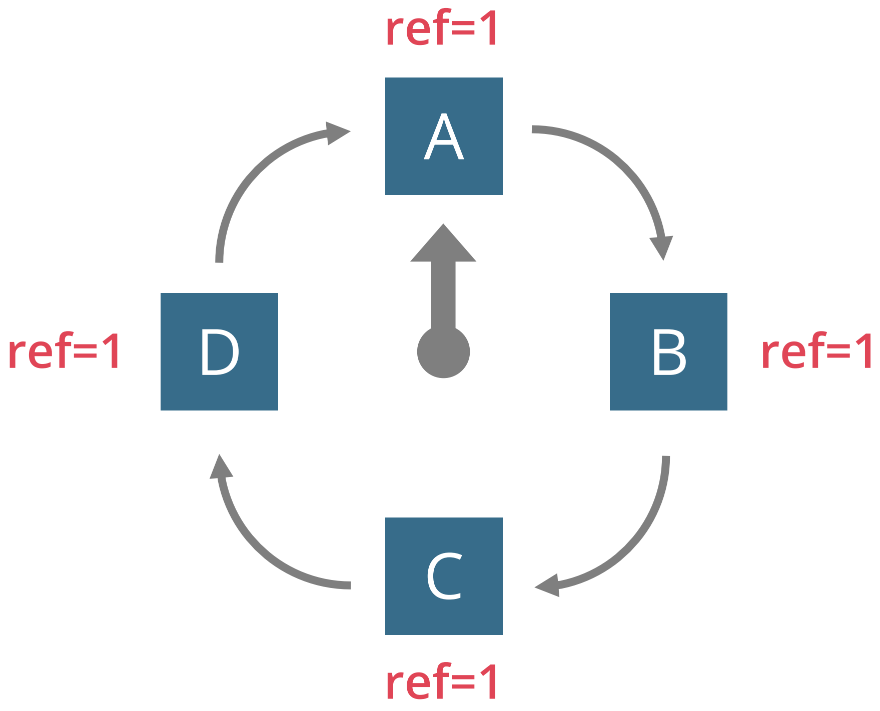
Page access sequence:

A B C D E B A D C A E C

Page E not present ⇒ buffer miss!

Find first frame with ref = 0

If ref = 1, unset it and move the hand




Buffer hits (so far) = 0

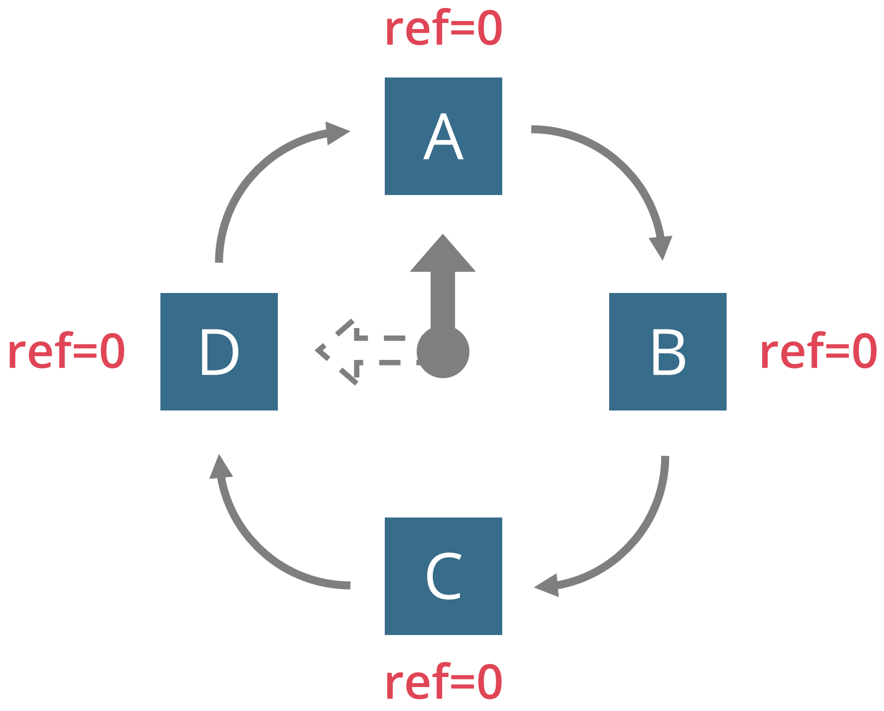
# Question 3, Part 4

Page access sequence:

A B C D E B A D C A E C

Resets bits of A, B, C, D while moving the hand

First frame with ref = 0 holds A




Buffer hits (so far) = 0

# Question 3, Part 5

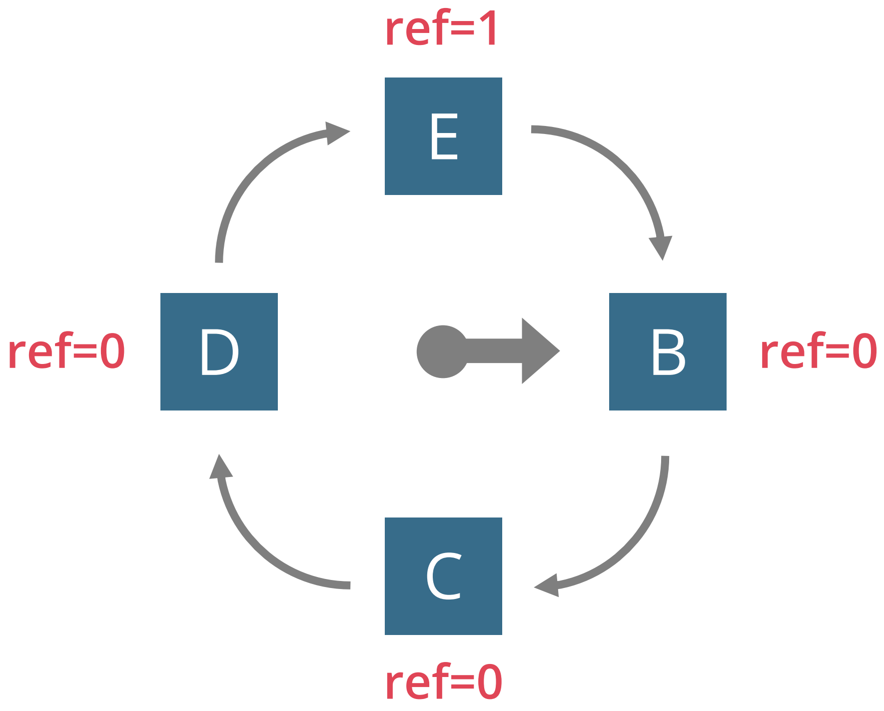
Page access sequence:

A B C D E B A D C A E C

Resets bits of A, B, C, D while moving the hand

First frame with ref = 0 holds A

Replace A with E, set reference bit, move the hand




Buffer hits (so far) = 0

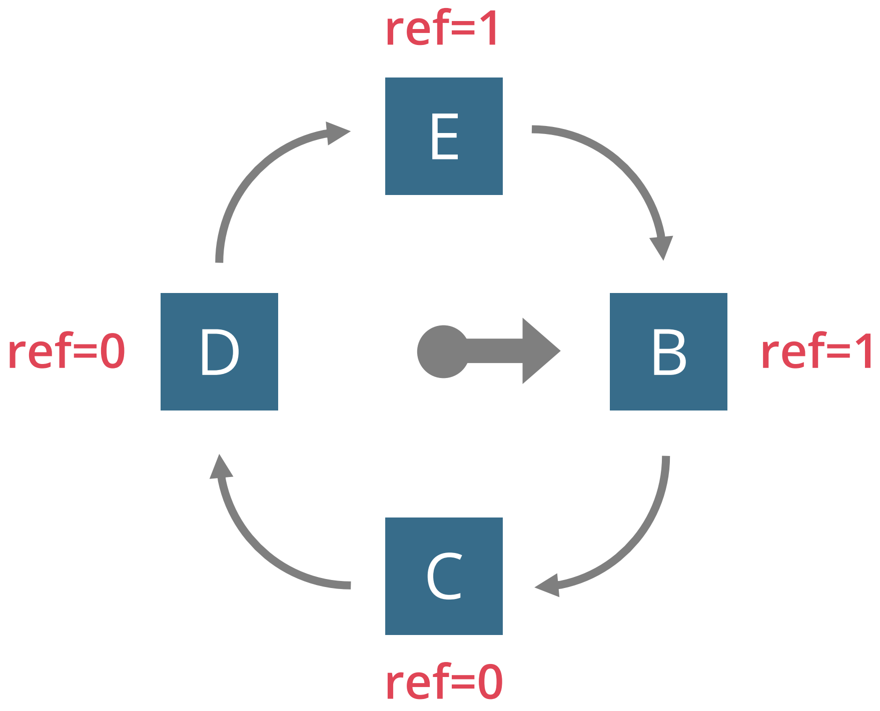
# Question 3, Part 6

Page access sequence:

A B C D E B A D C A E C

Page B is present ⇒ buffer hit!

Set refence bit



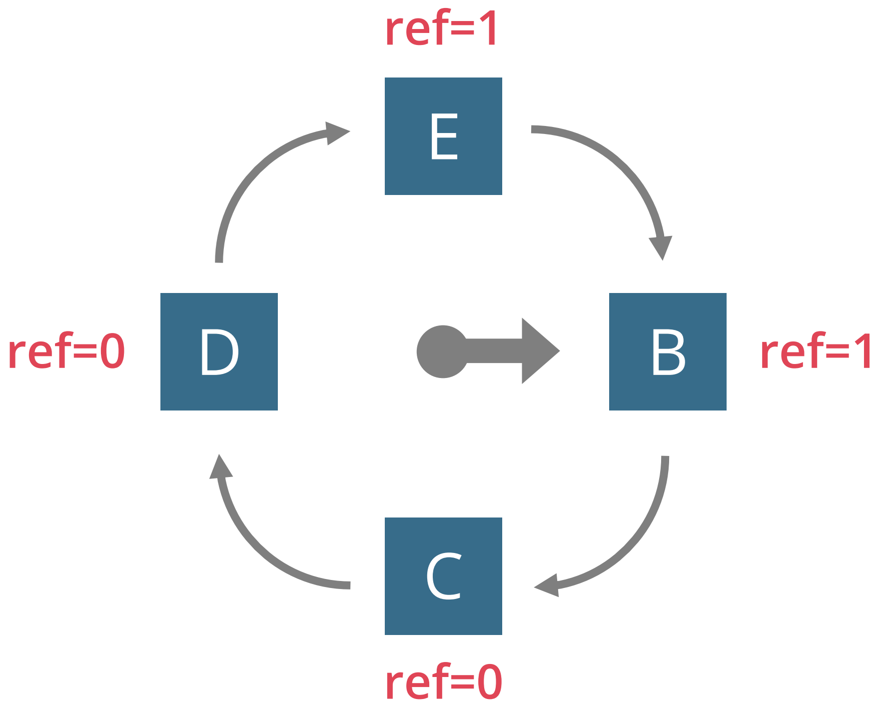

Buffer hits (so far) = 1

# Question 3, Part 7

Page access sequence:

A B C D E B A D C A E C

Page A not present ⇒ buffer miss!




Buffer hits (so far) = 1

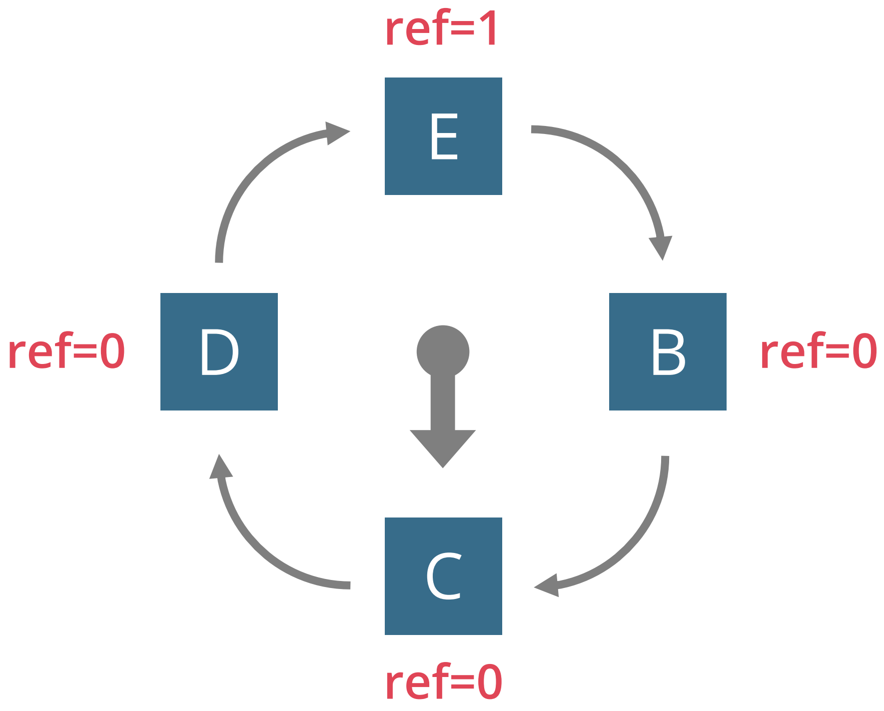
# Question 3, Part 8

Page access sequence:

A B C D E B A D C A E C

Page A not present ⇒ buffer miss!

Unset refence bit for B, move the hand




Buffer hits (so far) = 1

# Question 3, Part 9

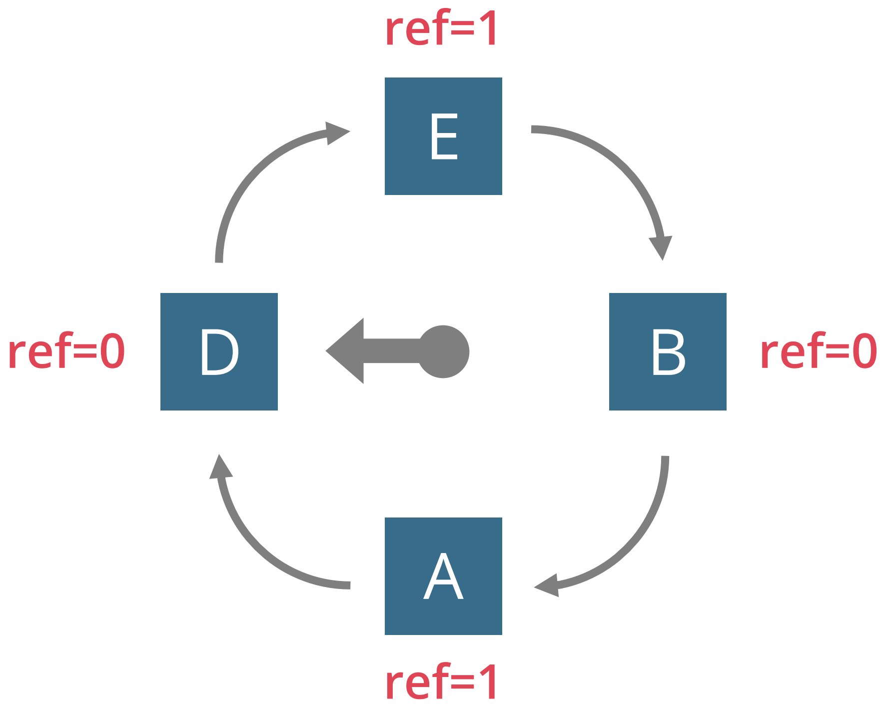
Page access sequence:

A B C D E B A D C A E C

Page A not present ⇒ buffer miss!

Unset refence bit for B, move the hand

Replace C with A, set refence bit, move the hand




Buffer hits (so far) = 1

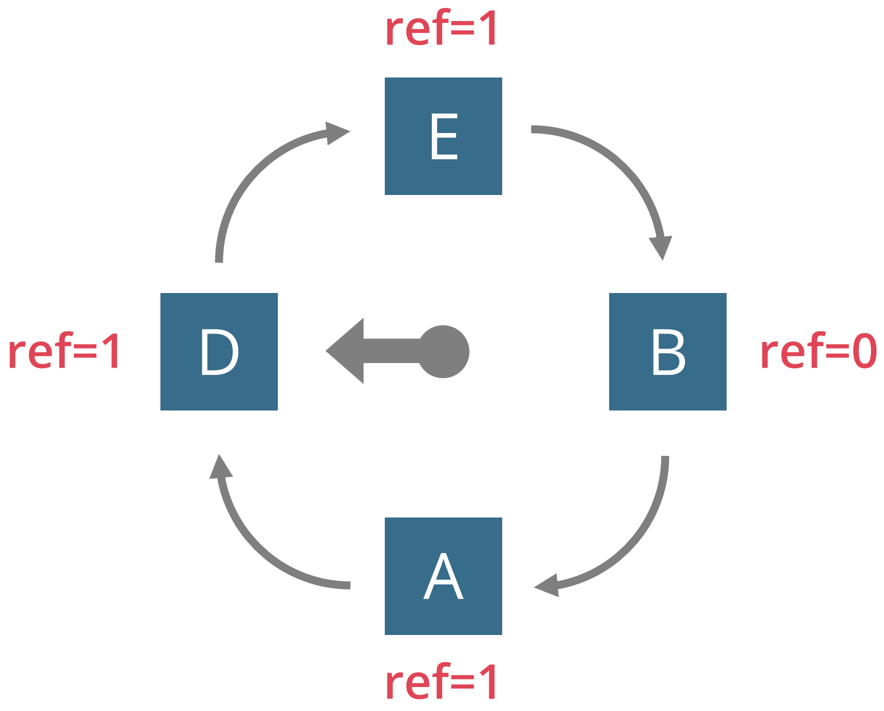
# QUESTION 3, PART 10

Page access sequence:

A B C D E B A D C A E C

Page D is present ⇒ buffer hit!

Set refence bit



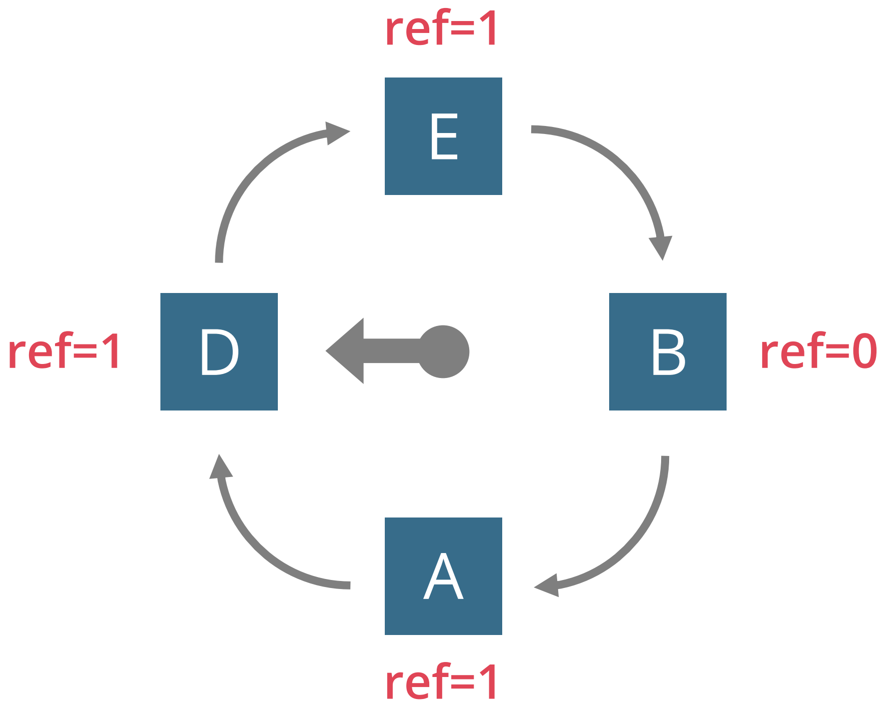

Buffer hits (so far) = 2

# Question 3, Part 11

Page access sequence:

A B C D E B A D C A E C

Page C is not present ⇒ buffer miss!




Buffer hits (so far) = 2

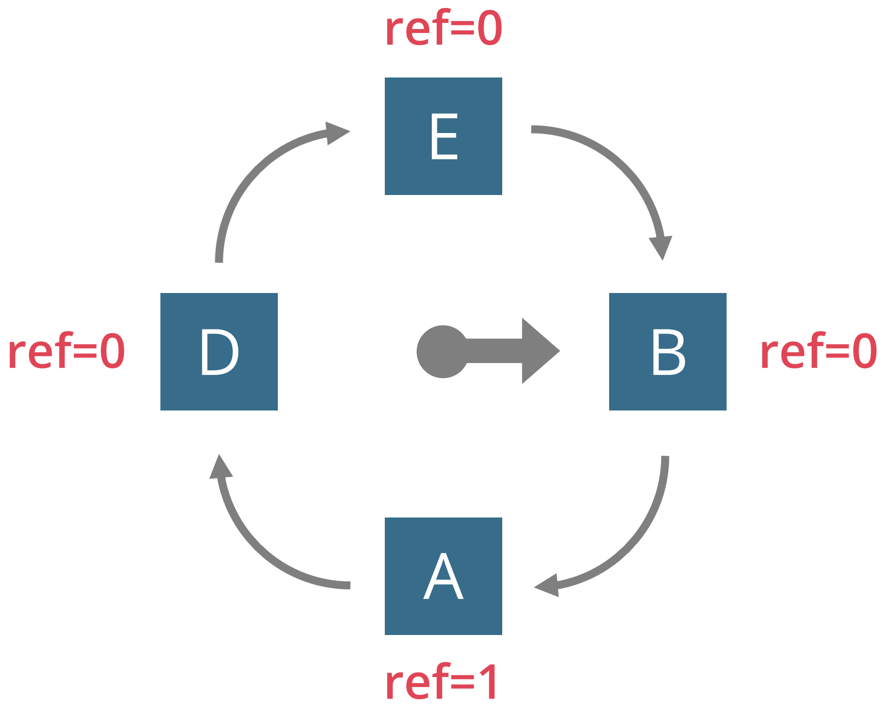
# Question 3, Part 12

Page access sequence:

A B C D E B A D C A E C

Page C is not present ⇒ buffer miss!

Unset ref bits for D & E, move the hand to B




Buffer hits (so far) = 2

# QUESTION 3, PART 13

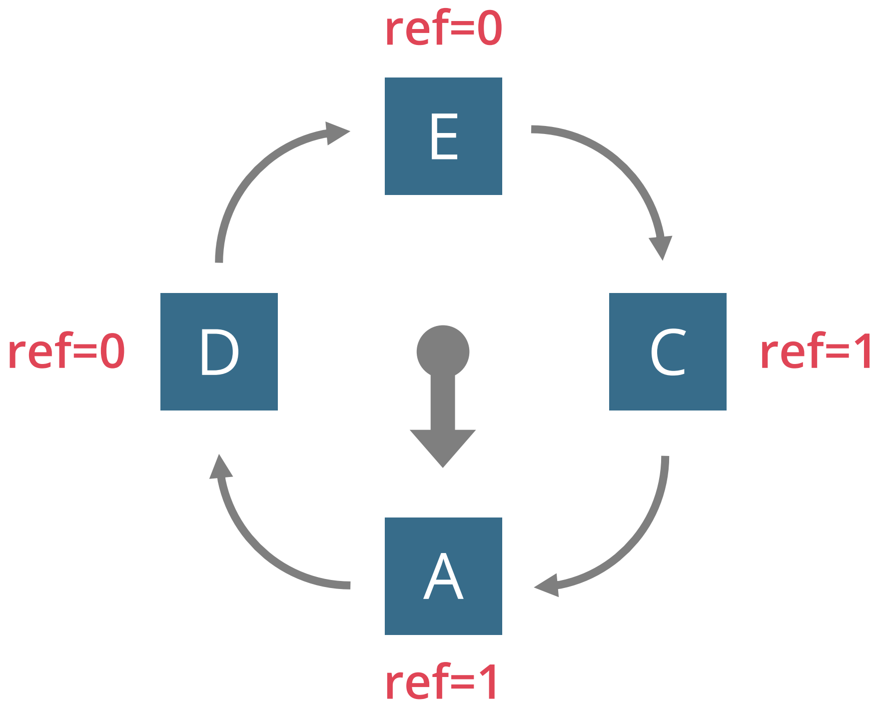
Page access sequence:

A B C D E B A D C A E C

Page C is not present ⇒ buffer miss!

Unset ref bits for D & E, move the hand to B

Replace B with C, set refence bit, move the hand



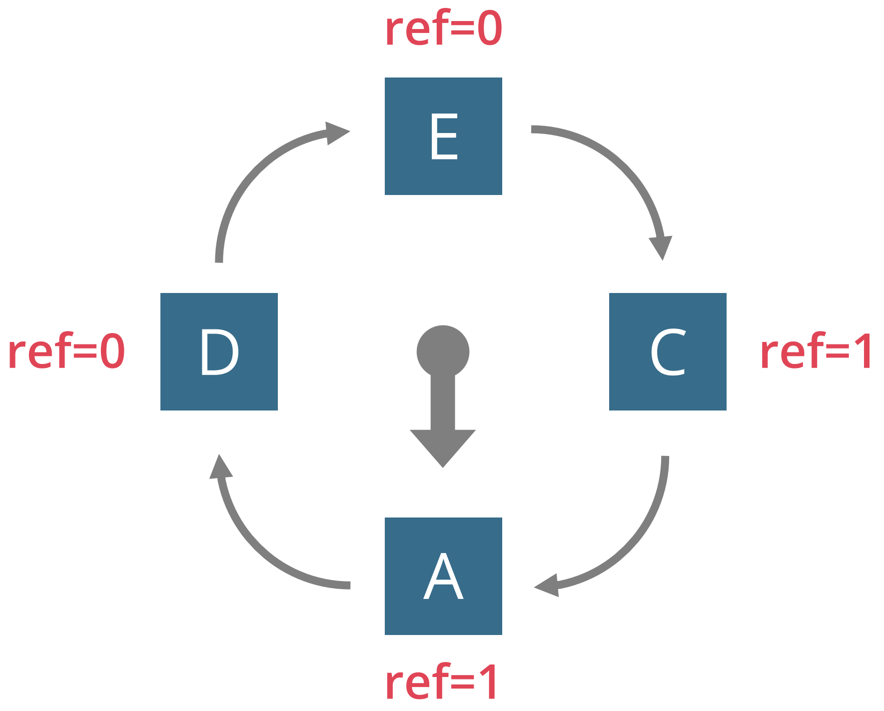

Buffer hits (so far) = 2

# Question 3, Part 14

Page access sequence:

A B C D E B A D C A E C

Pages A, E, C are present ⇒ buffer hits!




Buffer hits (so far) = 2

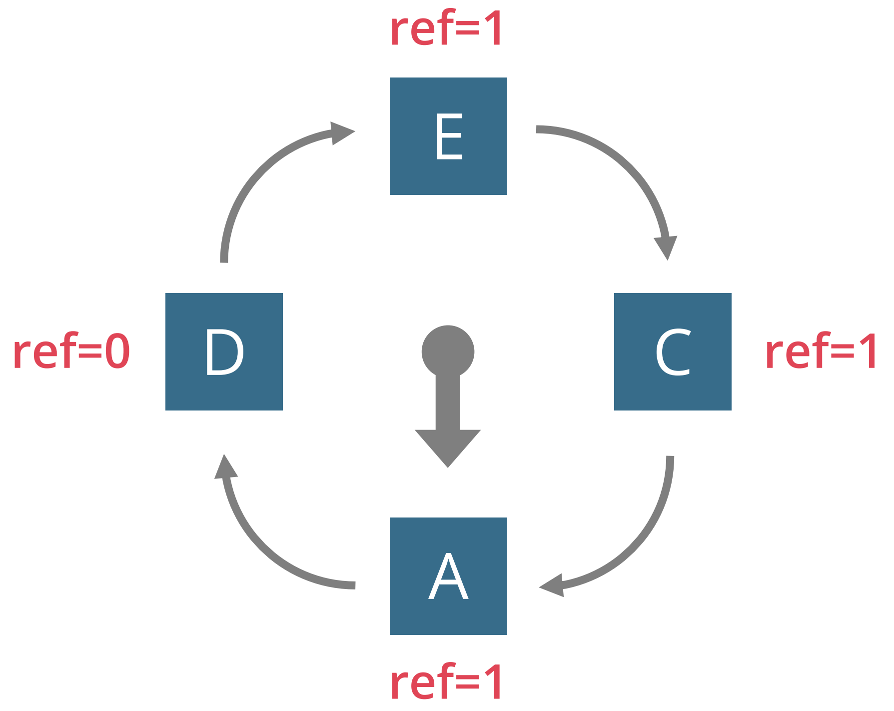
# Question 3, Part 15

Page access sequence:

A B C D E B A D C A E C

Pages A, E, C are present ⇒ buffer hits!

Set their reference bits




Buffer hits = 5

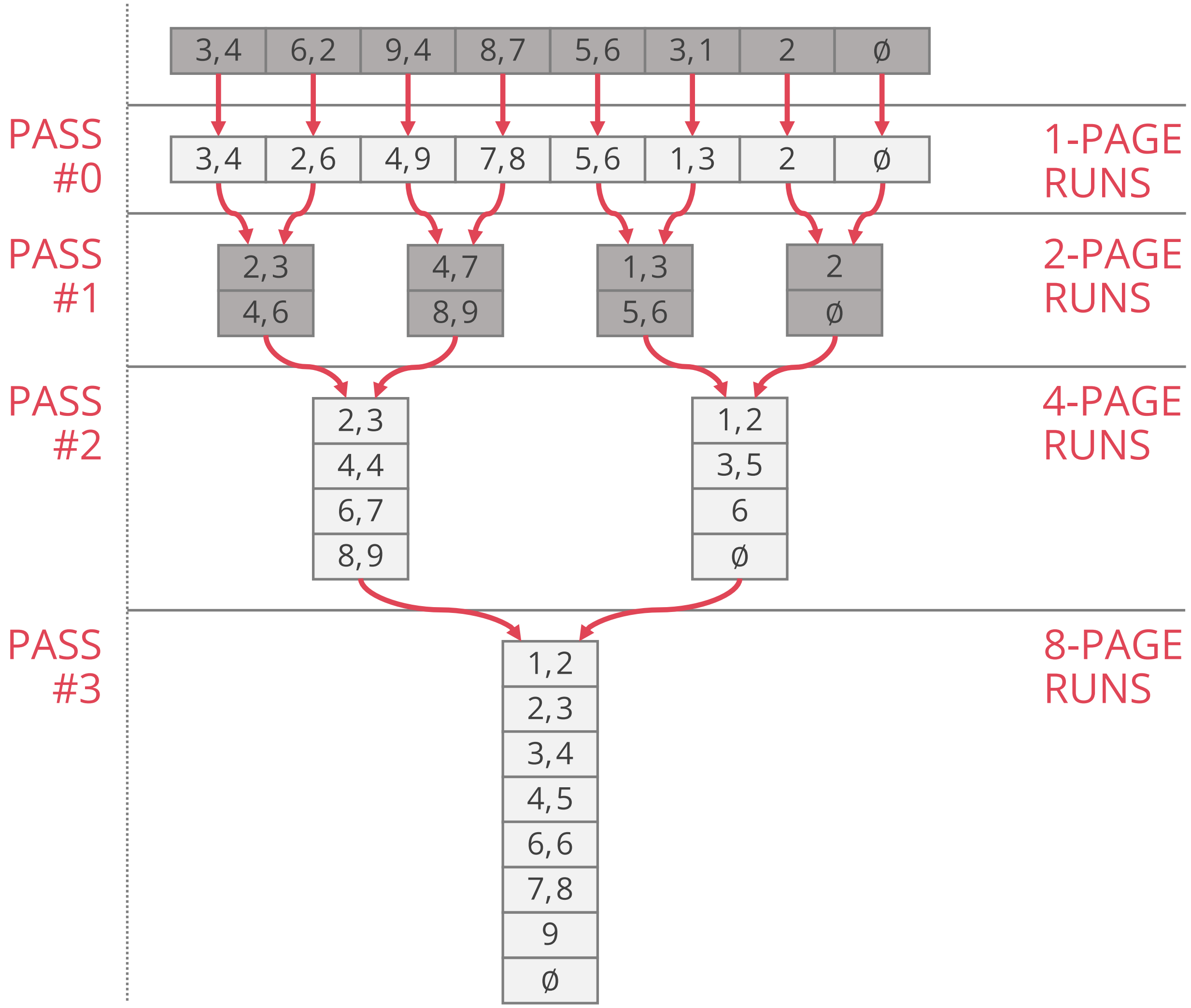
# Sorting

# SORTING

We first sort small amounts of data into **runs** of sorted tuples

Given runs of sorted tuples, we can **merge** them into 1 larger run of sorted tuples

- Same as in-memory merge sort
- Stream in the two runs and stream out the new run



# General External Merge Sort

## How many passes do we need?

We sort $B$ pages at once, so we have $\lceil N/B \rceil$ runs after Pass 0

We merge $B\text{-}1$ pages at once, so we have to do $\lceil \log_{B-1}(\#\ runs) \rceil$ merge passes

So we have $1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$ passes over the data

## I/O cost:

Read and write each page per pass

Total I/O cost = $2N \cdot (1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil)$

# General External Merge Sort

Number of passes = $1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$

How many pages can be sorted in $P$ passes?

- Two passes can sort $B \cdot (B-1)$ pages
- Three passes can sort $B \cdot (B-1)^2$ pages
- $P$ passes can sort $B \cdot (B-1)^{P-1}$ pages

# Question 4

Suppose the page size is 4 KB and the buffer pool size is 1 MB

How many I/Os are required to sort a relation of size 800 KB?

What is the size of the largest relation that would need two passes to sort?

# Question 4, Part 2

Suppose the page size is 4 KB and the buffer pool size is 1 MB

*B* = 1024KB / 4KB = 256 pages

How many I/Os are required to sort a relation of size 800 KB?

*N* = 800KB / 4KB = 200 pages

The relation can completely fit into the buffer, so we only need to read it in, sort it (no I/Os required for sorting), then write the sorted pages back to disk. Total: **400 I/Os**.

What is the size of the largest relation that would need two passes to sort?

Max number of pages: $B \cdot (B - 1) = 256 \cdot 255 = 65{,}280$

Max relation size = $65{,}280 \cdot 4$KB = 261,120KB

# Joins

# Nested Loops Joins

Simple / Page / Block Nested Loop Joins:

(all pages of left table) + (number of passes of right table) * (all pages of right table)

Number of passes:

Simple: one per left row

Page: one per left page

Block: one per left block

# Nested Loops Joins

Simple Nested Loops Join: $\text{pages}(R) + \text{tuples}(R) \cdot \text{pages}(S)$

Page Nested Loops Join: $\text{pages}(R) + \text{pages}(R) \cdot \text{pages}(S)$

Block Nested Loops Join: $\text{pages}(R) + \lceil \text{pages}(R) / (B - 2) \rceil \cdot \text{pages}(S)$

where $B$ is the number of available buffer pages

# Index Nested Loops Join

Index Nested Loop Join: **pages(R) + tuples(R) · cost to find matching S tuples**

(all pages of left table) + (number of right index lookups) · (cost of right index lookup)

Cost to find matching S tuples:

**Variant A**: just cost to traverse root to leaf + read all the leaves with matching tuples

**Variant B/C:** cost of retrieving RIDs (similar to Variant A) + cost to fetch actual tuples

1 I/O per **page** if clustered, 1 I/O per **tuple** if not

# Sort Merge Join

Sort Merge Join:

Cost to sort R using external sorting +

Cost to sort S using external sorting +

pages(R) + pages(S)

Note that, if a relation is already sorted, we can exclude that cost

# Sort Merge Join

Sort Merge Join optimisation: combine **last sort pass** with **merging**

Normally:

- Last sort pass:
  - Load runs R1, R2, R3 into buffers, merge into run R, stream (write) R to disk
  - Load runs S1,S2, S3 into buffers, merge into run S, stream (write) S to disk
- Merging:
  - Load run R and run S into buffers, merge into $R \bowtie S$

# Sort Merge Join

Sort Merge Join optimisation: combine **last sort pass** with **merging**

**Sort-merge join optimisation:**

Last sort pass:

Load runs R1, R2, R3 into buffers, ~~merge into run R, stream (write) R to disk~~

Load runs S1,S2, S3 into buffers, ~~merge into run S, stream (write) S to disk~~

Merging:

~~Load run R and run S into buffers~~, merge into R ⋈ S

Note that in this example, previously we needed only 3 input buffers, but the optimized version needed 6 input buffers!

# Sort Merge Join

Sort Merge Join optimisation: combine last sort pass with merging

**Sort-merge join optimisation:**

In general, this optimization is only possible if you happen to have enough buffers to stream **BOTH** last runs in memory

You can also do a partial version where you finish sorting one table normally, then do the join with the runs of the unmerged table and the one run of the merged table

You save $2 \cdot (\text{pages}(R) + \text{pages}(S))$ by doing this optimization

The partial version saves either $2 \cdot \text{pages}(R)$ or $2 \cdot \text{pages}(S)$, depending on which table you wait to merge

# Grace Hash Join

Grace Hash Join: similar to external hash, but…

- **Partitioning phase:** Partition ***R*** into ***B-1*** buckets and also ***S*** into ***B-1*** buckets
  - Recursively partition pairs of ***R*** and ***S*** partitions until one partition in a pair fits in ***B-2*** pages
- **Joining phase:** for each pair of partitions where at least one is at most ***B-2*** pages,
  - Load **smaller side** (e.g., R) into memory, and make a hash table
  - Stream in pages of S ⟶ match against hash table ⟶ stream out matches

Cost: Depends on the construction of the tables. It's similar to external hashing, but your parameters for stopping are different