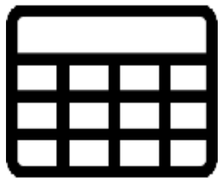
Tables Management



Database System Technology

Niv Dayan

We’ve decided to allow groups of both 2 and 3. Load is same. We recommend 3.

Please start forming groups and start step 1

Undergrads and grads can form groups

We enabled “Search for Teammates” on Piazza

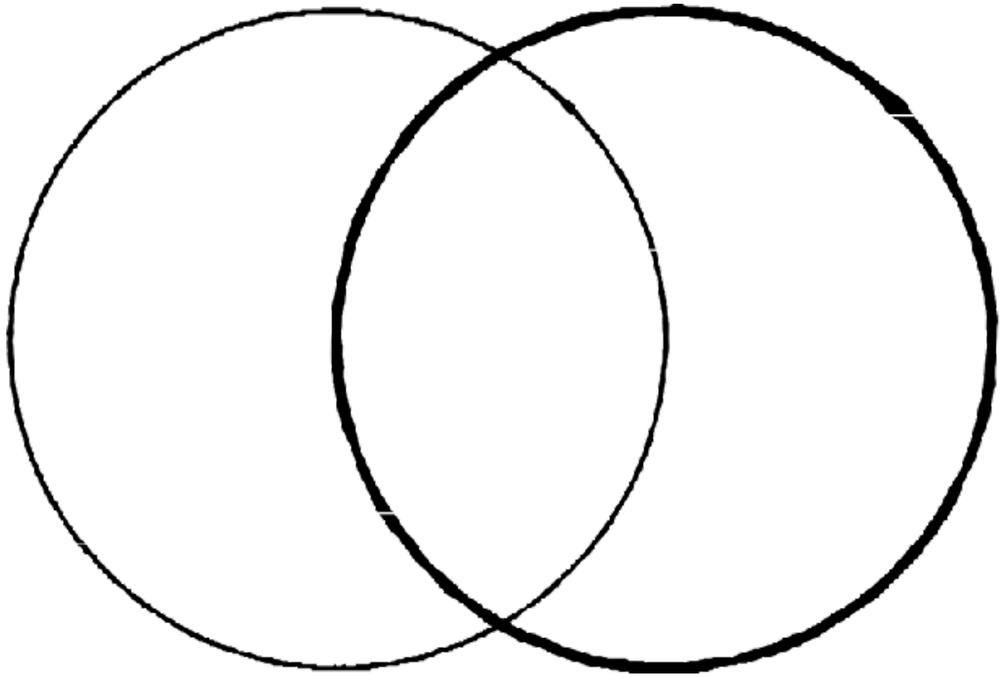
In person classes from this Thursday :)

This Thursday: Navid

There to solidify your understanding and get a historical perspective

Textbook

Slides



Only material in the slides will appear in the midterm/exam

Database Tables

A database consists of multiple tables

Customers

ID	Name	email	Addr

Orders

ID	Customer ID	Product ID	Date

Database Tables

A database consists of multiple tables

How do we store them in storage efficiently?

Customers

ID	Name	email	Addr

Orders

ID	Customer ID	Product ID	Date

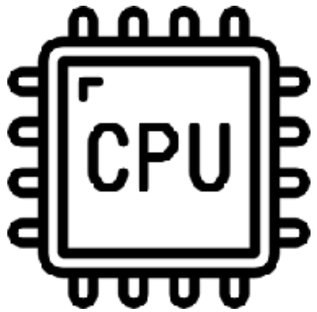
Operations to Efficiently Support

1. Scans — e.g., select * from Customers
2. Deletes — e.g., delete from Customers where name = “…”
3. Updates — e.g., update Customers set email = “…” where name = “”
4. Insertions — e.g., Insert into Customers (, , ,)

Customers

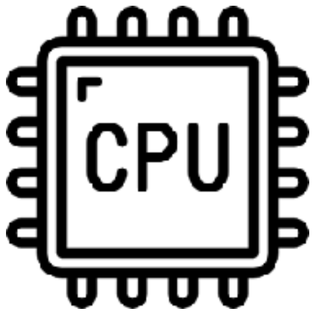
ID	Name	email	Addr

Optimizing for Data Movement

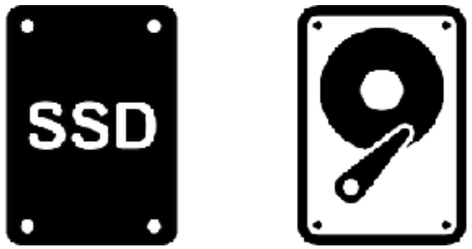


In previous courses on algorithms & data structures, you learned to optimize CPU cycles for an algorithm.

Optimizing for Data Movement



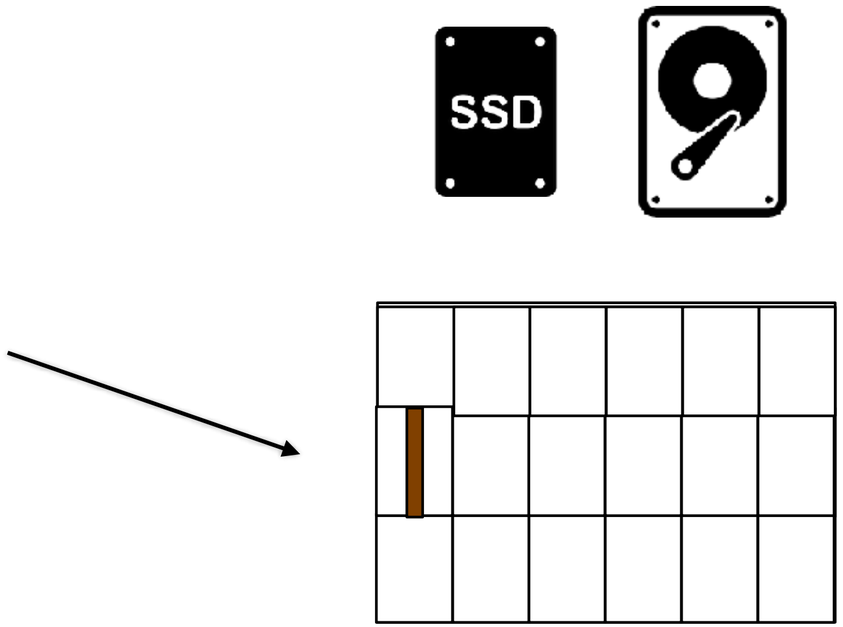
In previous courses on algorithms & data structures, you learned to optimize CPU cycles for an algorithm.



As storage devices are far slower, in this course we focus on optimizing data movement.

First Insight: Database Pages

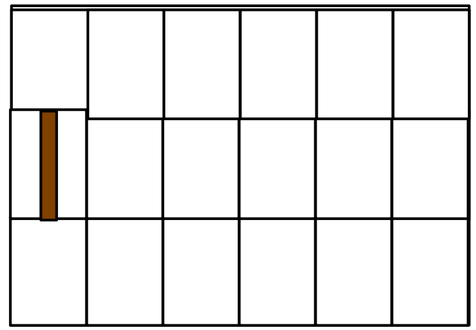
Reading/writing from storage at units of less than ≈4KB does not pay off.



First Insight: Database Pages

Reading/writing from storage at units of less than ≈4KB does not pay off.

Why? (Different reasons for disk and SSDs)

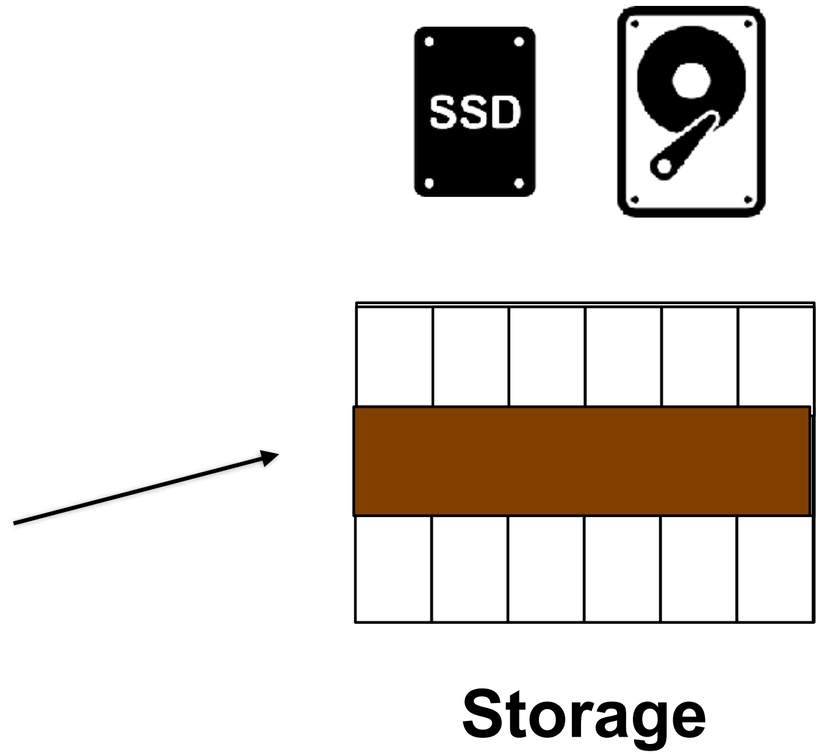


Storage

First Insight: Database Pages

Reading/writing from storage at units of less than ≈4KB does not pay off.

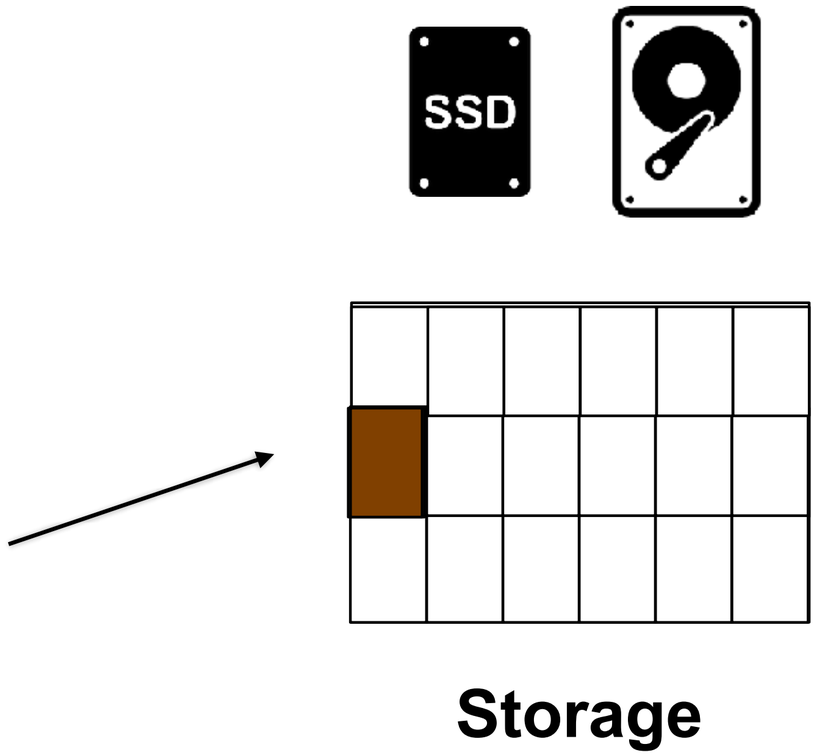
Reading/writing at very large units consumes memory and is less flexible for applications



First Insight: Database Pages

To balance, DBs use ≈4KB as the read/write unit. This is known as a database page.

An I/O (input/output) is one read or write request of one database page.



The Disk Access Model (The DAM Model)

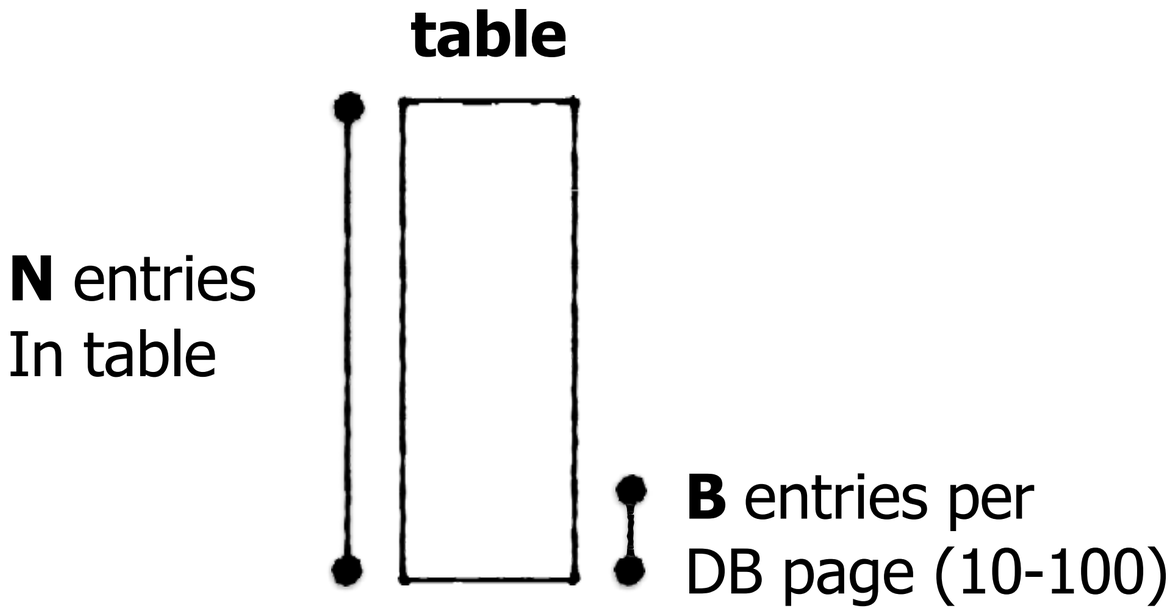
We will shortly propose algorithms to support scans/delete/updates/inserts

To reason about such algorithms, we need a cost model

The Disk Access Model (The DAM Model)

We will shortly propose algorithms to support scans/delete/updates/inserts

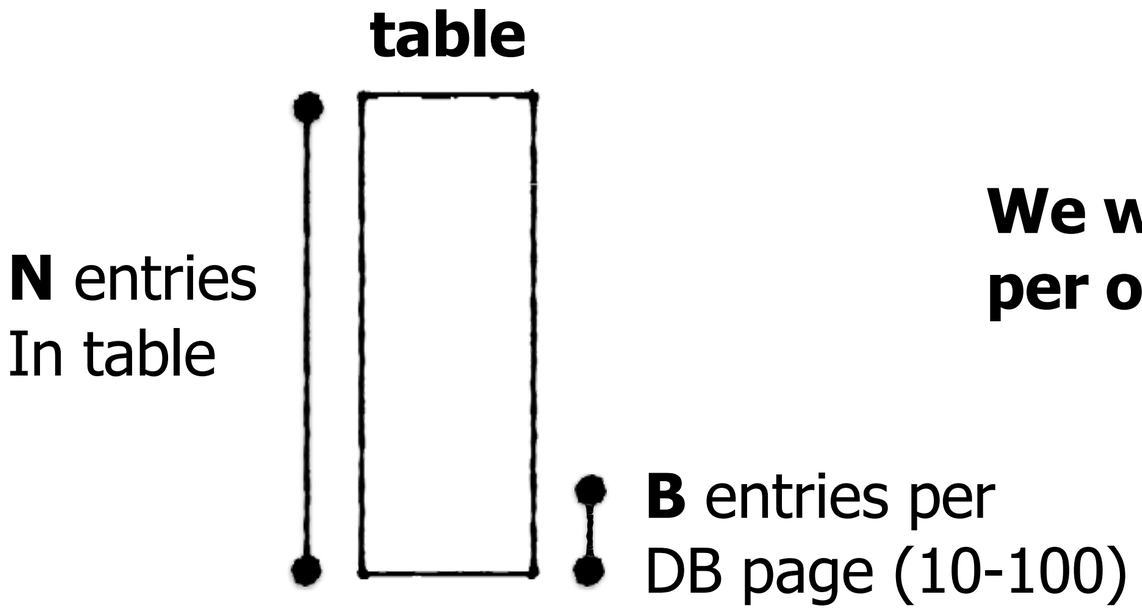
To reason about such algorithms, we need a cost model



The Disk Access Model (The DAM Model)

We will shortly propose algorithms to support scans/delete/updates/inserts

To reason about such algorithms, we need a cost model



We will count the worst-case number of I/Os per operations with respect to N and B

The Disk Access Model (The DAM Model)

This model is imperfect. It ignores many characteristics of storage.

The Disk Access Model (The DAM Model)

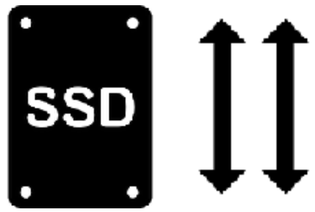
This model is imperfect. It ignores many characteristics of storage.

Ignores that sequential access is faster than random on disk

The Disk Access Model (The DAM Model)

This model is imperfect. It ignores many characteristics of storage.

Ignores that sequential access is faster than random on disk

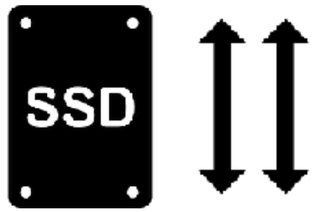


Ignores that SSD asynchronous I/O are faster

The Disk Access Model (The DAM Model)

This model is imperfect. It ignores many characteristics of storage.

Ignores that sequential access is faster than random on disk



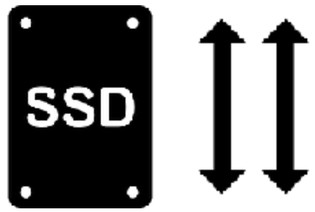
Ignores that SSD asynchronous I/O are faster

Ignores SSD garbage-collection due to random writes

The Disk Access Model (The DAM Model)

This model is imperfect. It ignores many characteristics of storage.

Ignores that sequential access is faster than random on disk



Ignores that SSD asynchronous I/O are faster

Ignores SSD garbage-collection due to random writes

However, it's useful due to its simplicity.

Operations

1. Scans e.g., select * from Customers
2. Deletes e.g., delete from Customers where name = “…”
3. Updates e.g., update Customers set email = “…” where name = “”
4. Insertions e.g., Insert into Customers (, , ,)

Scans - How not to Support Them

Customers

ID	Name	email	Addr

Orders

ID	Customer ID	Product ID	Date

Address Space

Scans - How not to Support Them

Customers

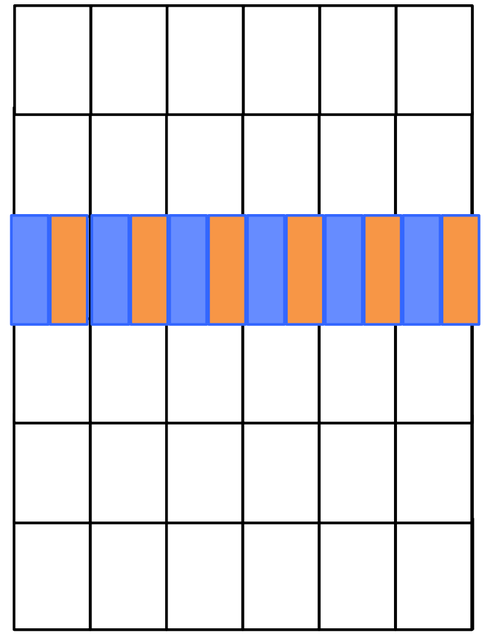
ID	Name	email	Addr

Orders

ID	Customer ID	Product ID	Date

Mix table rows within the same pages

Address Space



Scan cost?

Scans - How not to Support Them

Customers

N rows

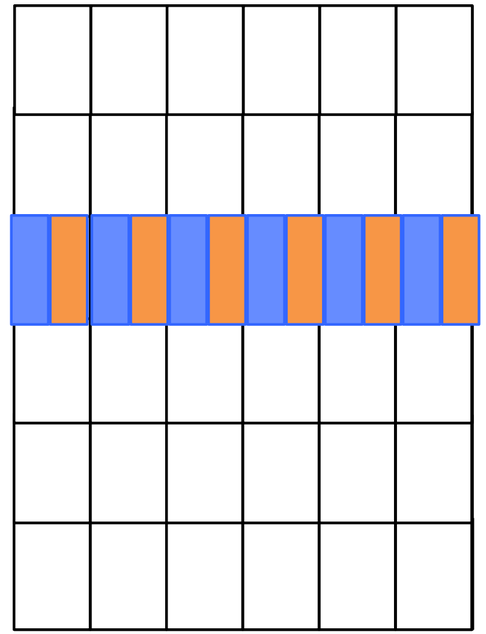
ID	Name	email	Addr

Orders

ID	Customer ID	Product ID	Date

Mix table rows within the same pages

Address Space



Scan cost: O(N) I/Os

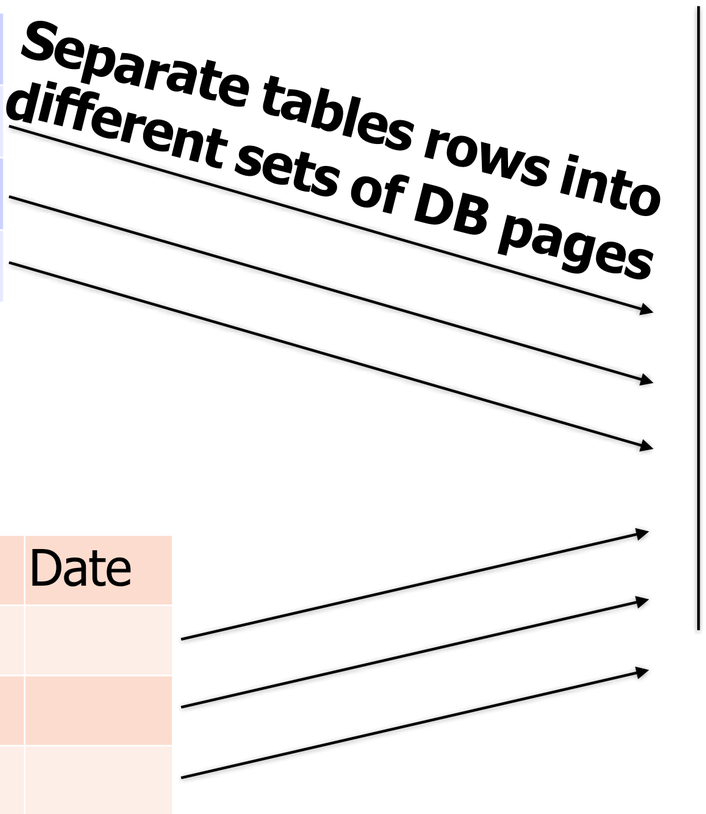
Efficient Scans

Customers

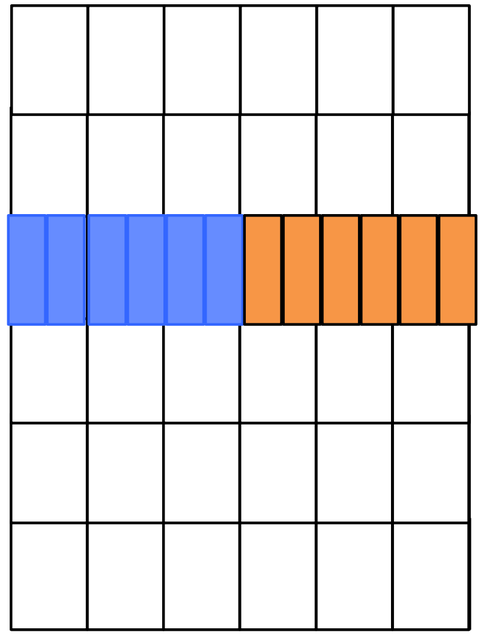
ID	Name	email	Addr

Orders

ID	Customer ID	Product ID	Date



Address Space



Scan cost?

Efficient Scans

Customers

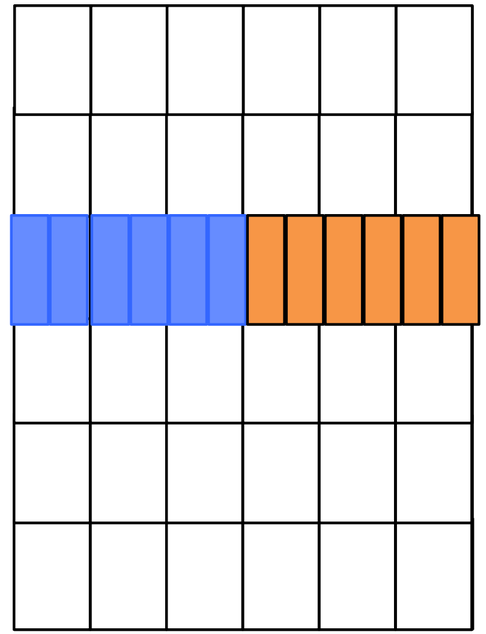
ID	Name	email	Addr

Orders

ID	Customer ID	Product ID	Date

Separate tables rows into different sets of DB pages

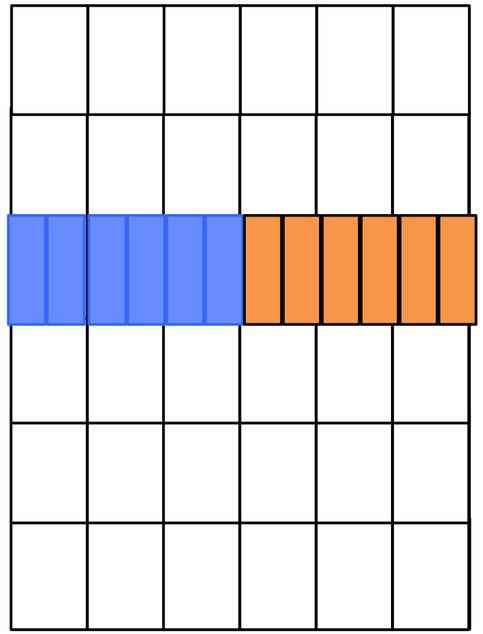
Address Space



Scan cost: O(N/B) I/Os

Efficient Scans

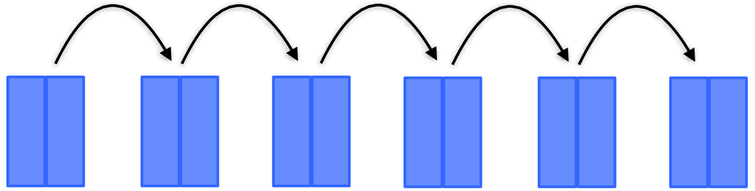
Which pages belong to which table?



Efficient Scans

Which pages belong to which table?

Simplest Solution: Linked List

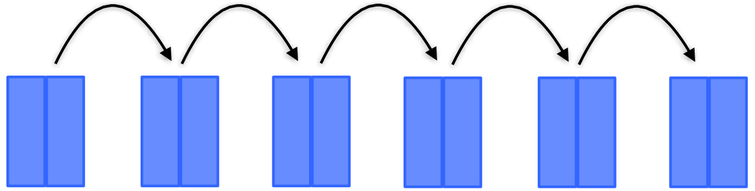


Efficient Scans

Which pages belong to which table?

Simplest Solution: Linked List

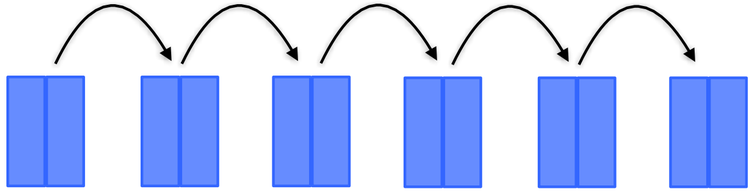
Problem?



Efficient Scans

Which pages belong to which table?

Simplest Solution: Linked List



Problem: **entails synchronous I/Os, which do not exploit SSD parallelism**

Solution:

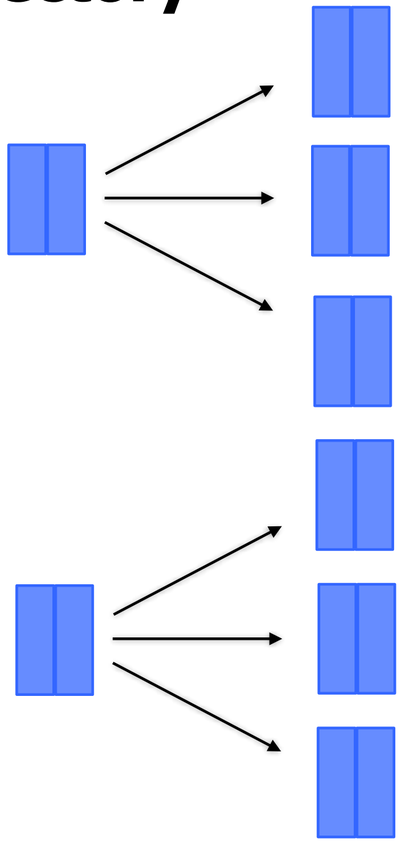
Efficient Scans

Which pages belong to which table?

Simplest Solution: Linked List

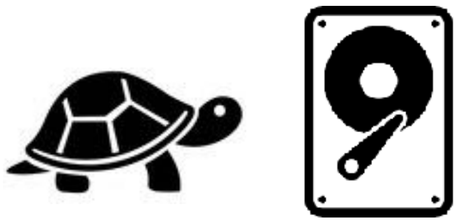
Problem: entails synchronous I/Os, which do not exploit SSD parallelism

Solution: **Employ directory to allow reading many pages asynchronously**

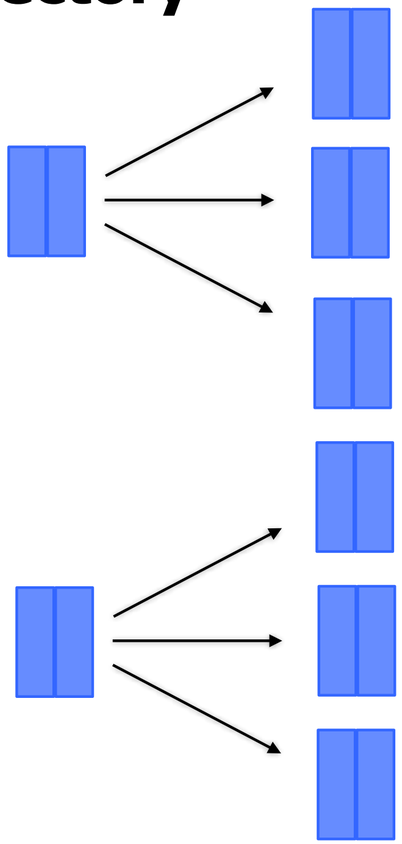


Efficient Scans

Problem for disk?

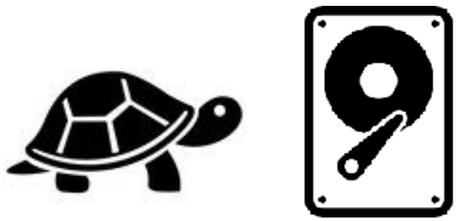


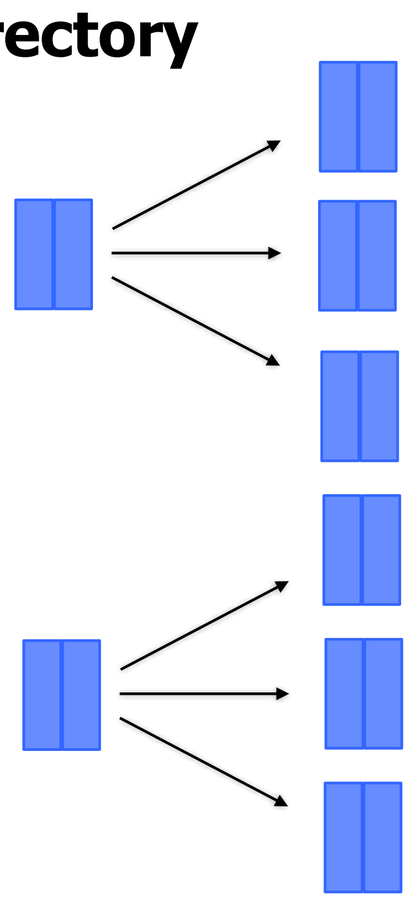
Directory



Efficient Scans

Problem: small I/Os, which do not saturate a disks's sequential bandwidth



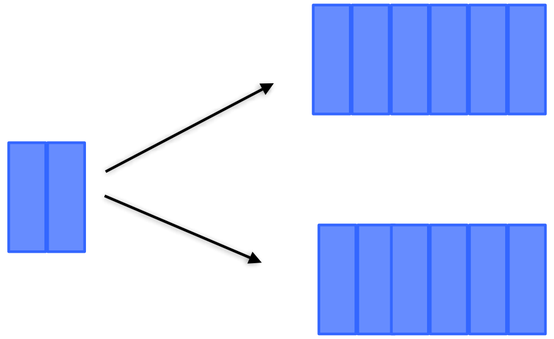


Efficient Scans

Problem: small I/Os, which do not saturate a disks's sequential bandwidth

Solution: **Store multiple database pages contiguously along "extents"** (8-64 pages)

Directory



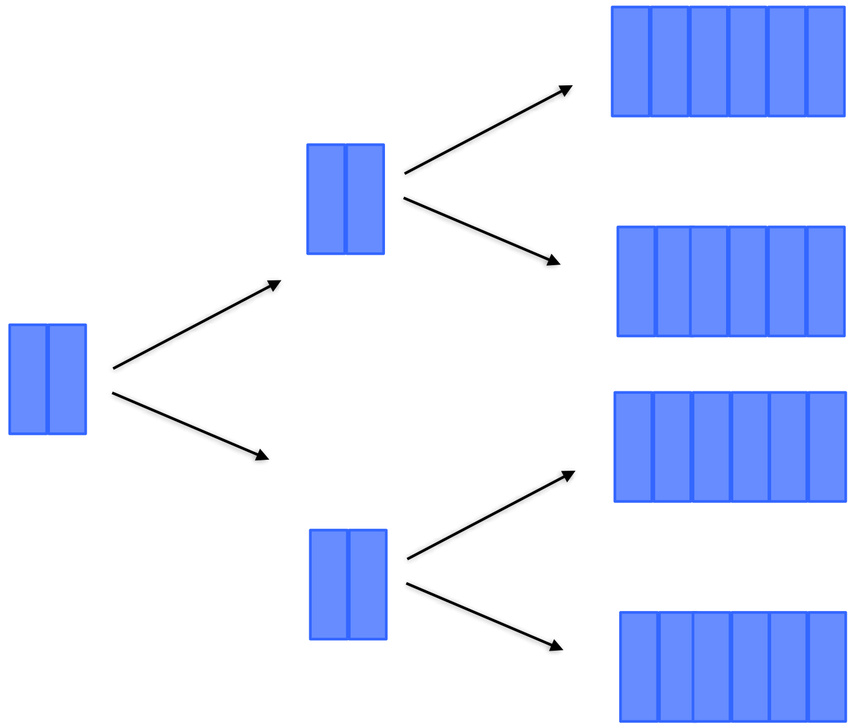
Efficient Scans

Problem: small I/Os, which do not saturate a disks's sequential bandwidth

Solution: Store multiple database pages contiguously along "extents" (8-64 pages)

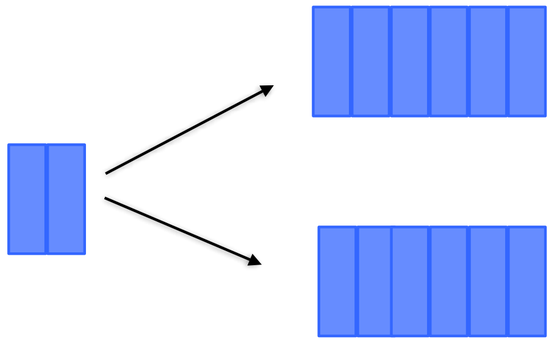
Bonus: Saves some metadata

File can grow as a tree if it gets large



Efficient Scans

How to keep track of directories of all files?

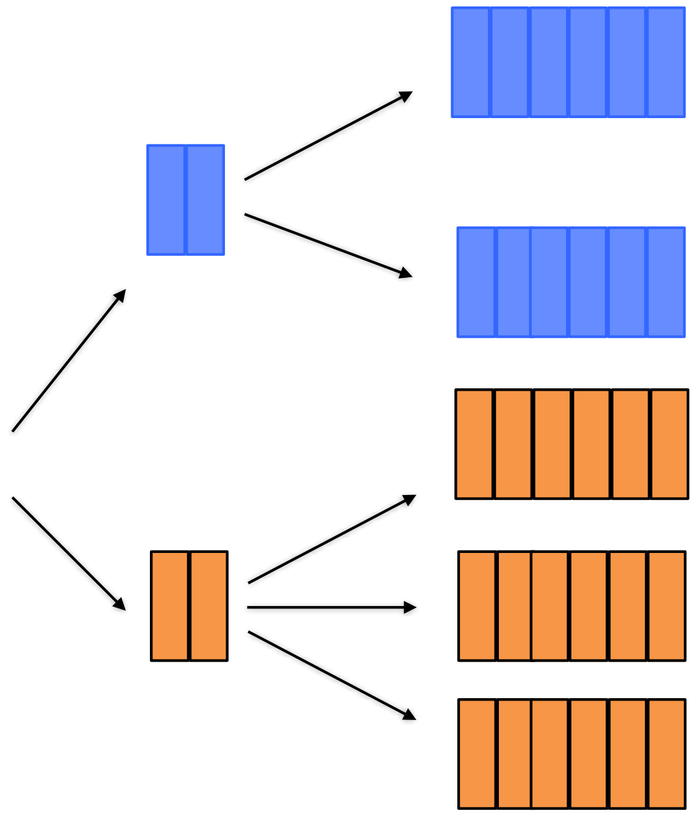


Efficient Scans

How to keep track of directories of all files?

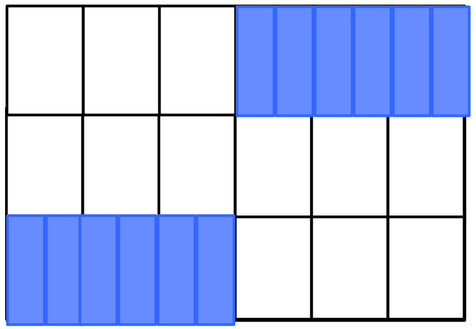
Catalog

Name	Data type	Size (Bytes)	Start
Customers			
Orders			



Efficient Scans

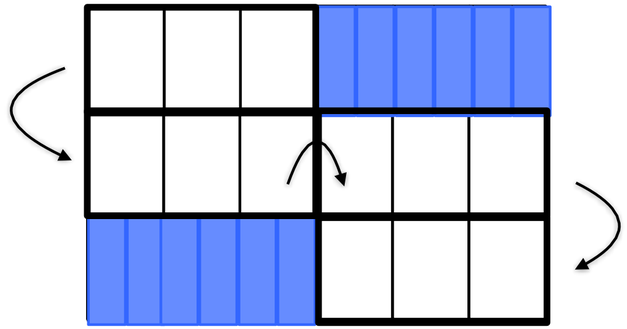
How to keep track of free pages/extents?



Efficient Scans

How to keep track of free pages/extents?

Solution 1: linked list (slower)

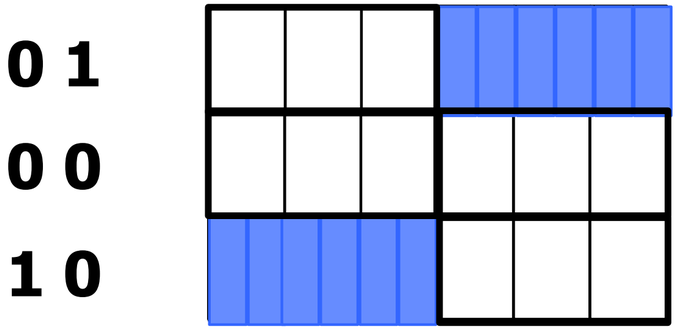


Efficient Scans

How to keep track of free pages/extents?

Solution 1: linked list (slower)

Solution 2: bitmap (takes space)



Operations

1. Scans e.g., select * from Customers
2. **Deletes** **e.g., delete from Customers where name = “…”**
3. Updates e.g., update Customers set email = “…” where name = “”
4. Insertions e.g., Insert into Customers (, , ,)

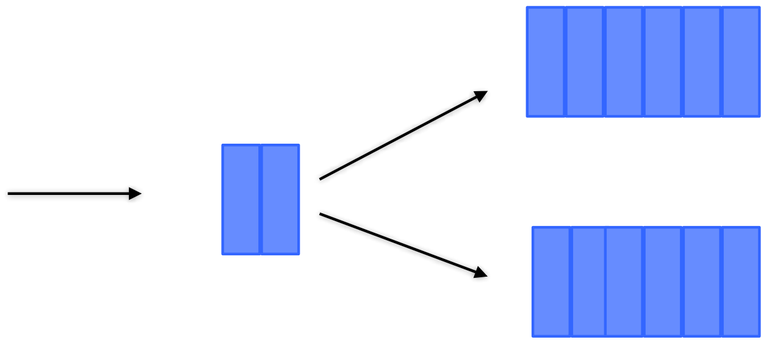
Supporting Deletes

e.g., delete from Customers where name = "…"

Simplest solution?

Catalog

Name	Data type	Size (Bytes)	Start
Customers			
Orders			



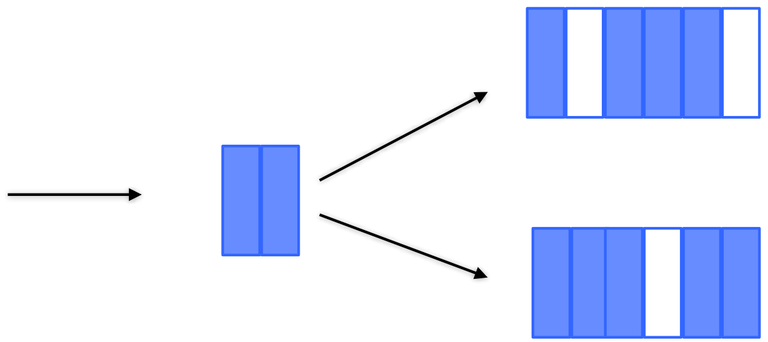
Supporting Deletes

e.g., delete from Customers where name = "…"

Simplest solution? **Scan the table. Create "holes".**

Catalog

Name	Data type	Size (Bytes)	Start
Customers			
Orders			



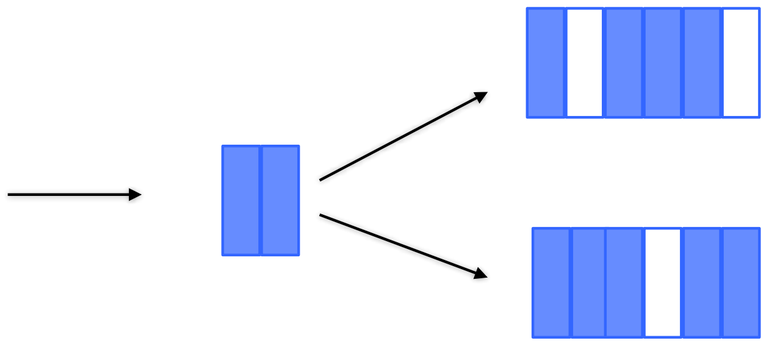
Supporting Deletes

e.g., delete from Customers where name = "…"

Simplest solution? Scan of the table. Creates "holes".

Catalog

Name	Data type	Size (Bytes)	Start
Customers			
Orders			



Cost: O(1) write and O(N/B) reads.

Operations

1. Scans
2. Deletes
3. **Updates**
4. Insertions

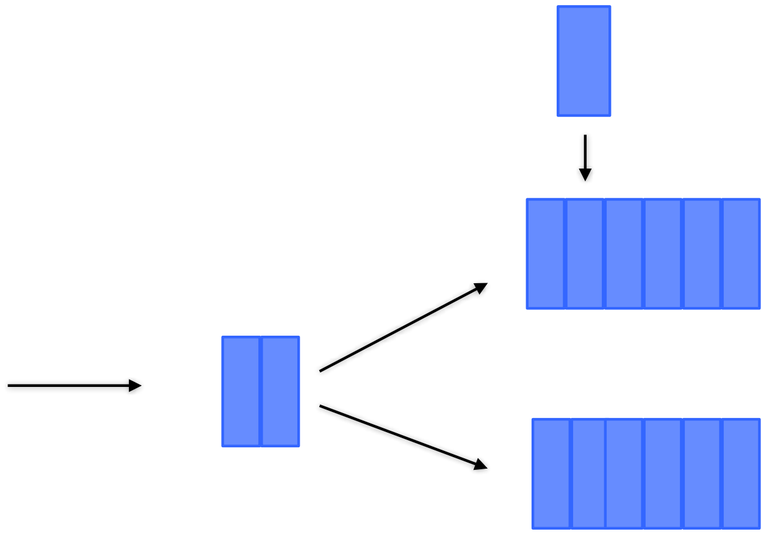
Supporting Updates

e.g., update Customers set email = "…" where name = "”

Scan and update.

Catalog

Name	Data type	Size (Bytes)	Start
Customers			
Orders			



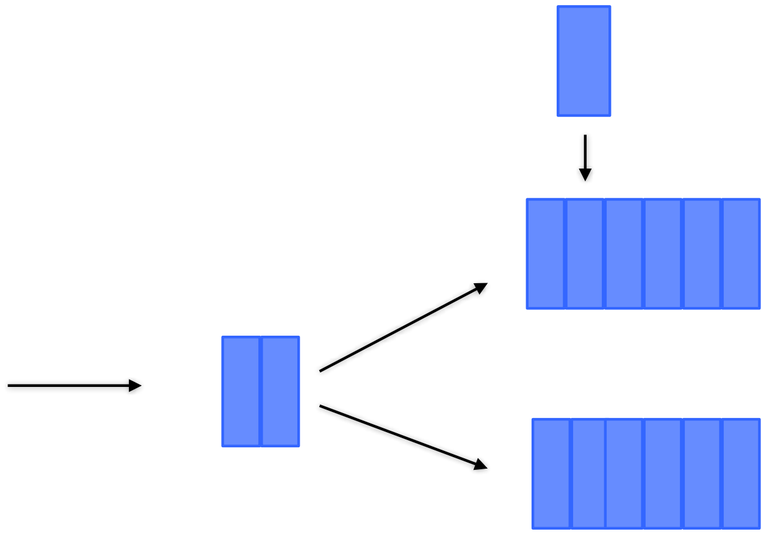
Supporting Updates

e.g., update Customers set email = "…" where name = "”

Scan and update. If newer version is too large, delete & reinsert

Catalog

Name	Data type	Size (Bytes)	Start
Customers			
Orders			



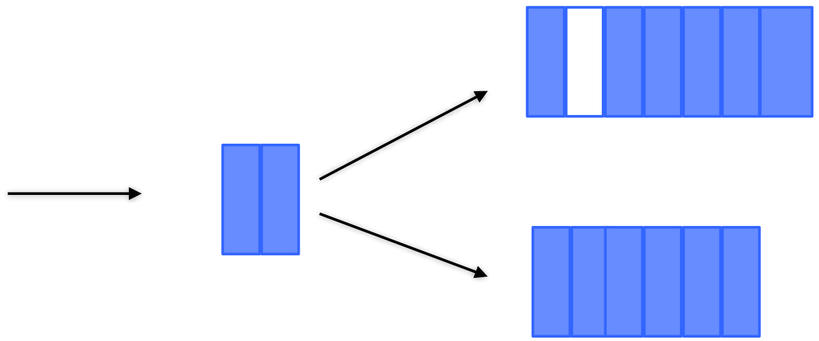
Supporting Updates

e.g., update Customers set email = "…" where name = "”

Scan and update. If newer version is too large, delete & reinsert

Catalog

Name	Data type	Size (Bytes)	Start
Customers			
Orders			



Cost: O(1) write and O(N/B) reads

Operations

1. Scans
2. Deletes
3. Updates
4. **Insertions**

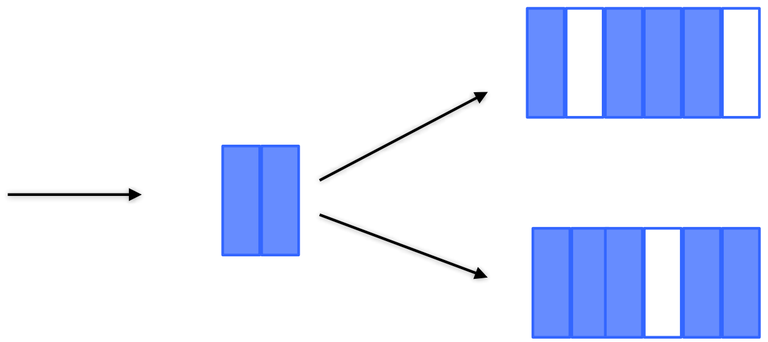
Supporting Insertions

e.g., Insert into Customers (, , ,)

Solutions?

Catalog

Name	Data type	Size (Bytes)	Start
Customers			
Orders			

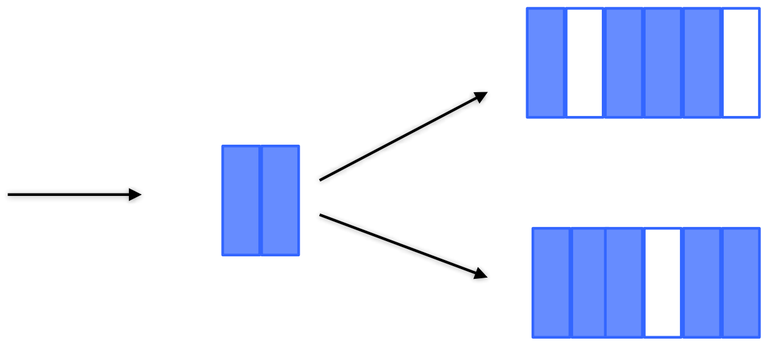


Supporting Insertions

(1) Scan & find space. Cost: O(N/B) reads and O(1) write.

Catalog

Name	Data type	Size (Bytes)	Start
Customers			
Orders			



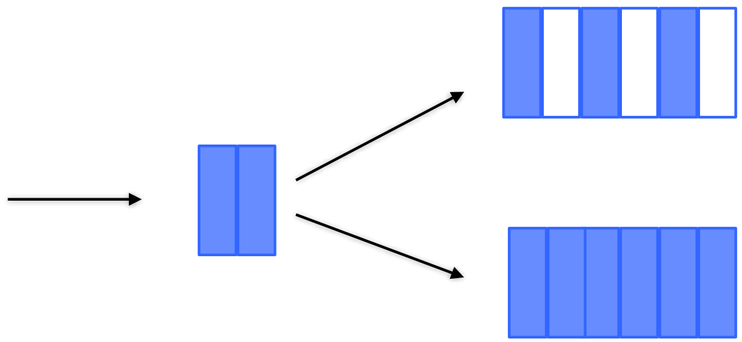
Supporting Insertions

(1) Scan & find space. Cost: O(N/B) reads and O(1) write.

(2) Separate Linked list of pages with free space.

Catalog

Name	Data type	Size (Bytes)	Start
Customers			
Orders			



Supporting Insertions

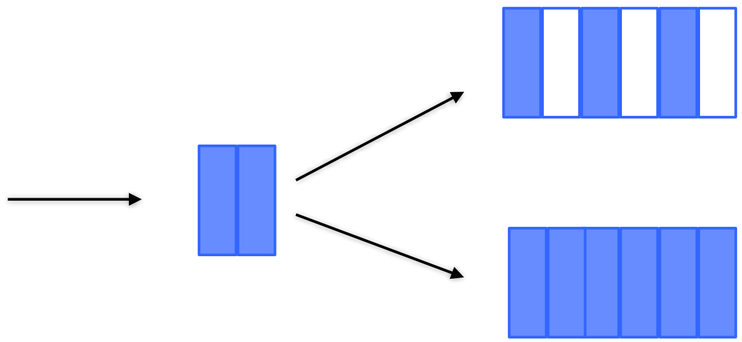
(1) Scan & find space. Cost: O(N/B) reads and O(1) write.

(2) Separate Linked list of pages with free space.

Cost: O(1) reads & O(1) write for fixed-sized entries

Catalog

Name	Data type	Size (Bytes)	Start
Customers			
Orders			



Supporting Insertions

(1) Scan & find space. Cost: O(N/B) reads and O(1) write.

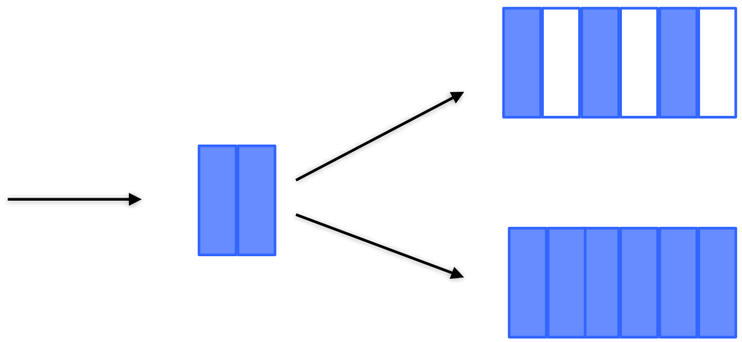
(2) Separate Linked list of pages with free space.

Cost: O(1) reads & O(1) write for fixed-sized entries

Cost: O(N/B) reads & O(1) write for variable-sized entries

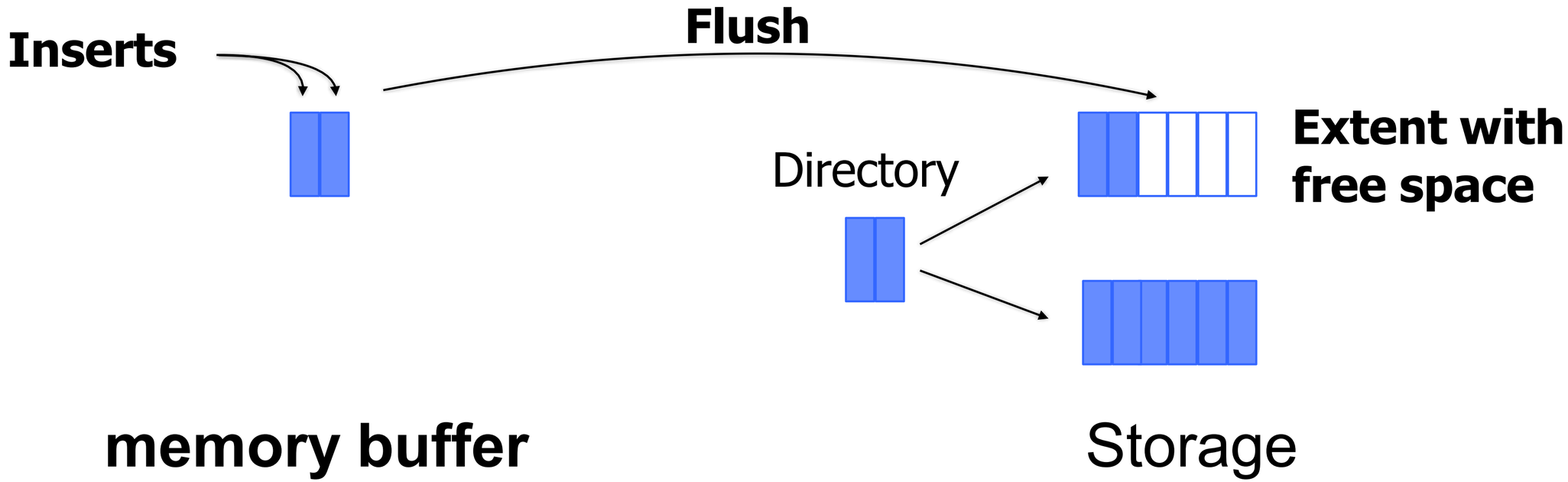
Catalog

Name	Data type	Size (Bytes)	Start
Customers			
Orders			



Supporting Insertions

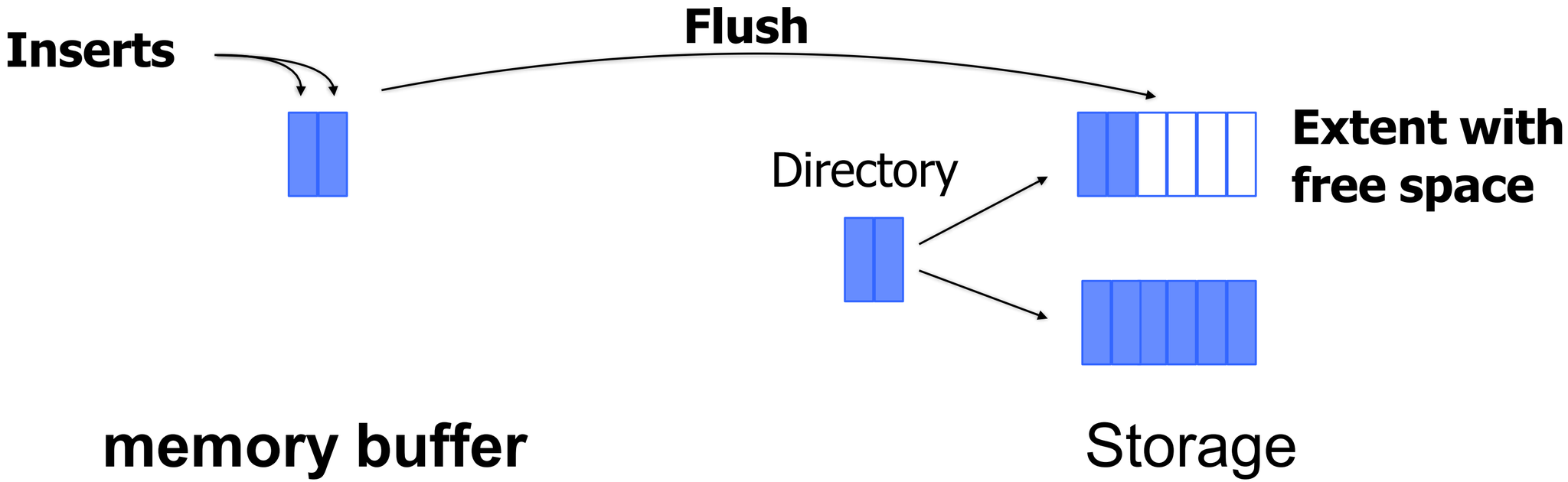
(3) buffer insertions in memory until a page fills up & append to extent



Supporting Insertions

(3) buffer insertions in memory until a page fills up & append to extent

Cost: No reads and O(1/B) of a write



Supporting Insertions

(1) Scan & find space. Cost: O(N/B) reads and O(1) write.

(2) Separate Linked list of pages with free space.
Cost: O(1) reads & O(1) write for fixed-sized entries Cost:
O(N/B) reads & O(1) write for variable-sized entries

(3) buffer insertions in memory until a page fills up & append to extent
Cost: No reads and O(1/B) of a write

Internal Page Organization

Recall each page is 4 KB

Suppose rows are fixed-sized

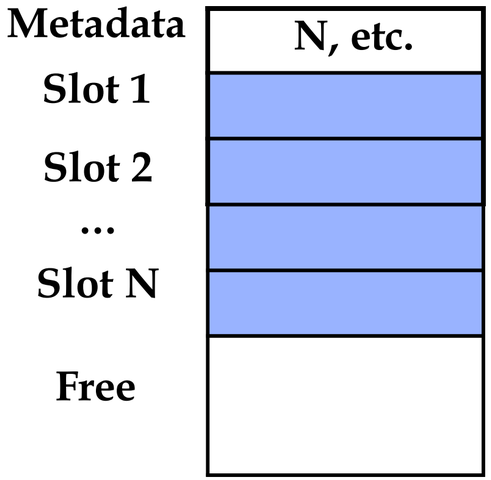
How to organize rows within a slot?

Internal Page Organization

Recall each page is 4 KB

Suppose rows are fixed-sized

How to organize rows within a slot?

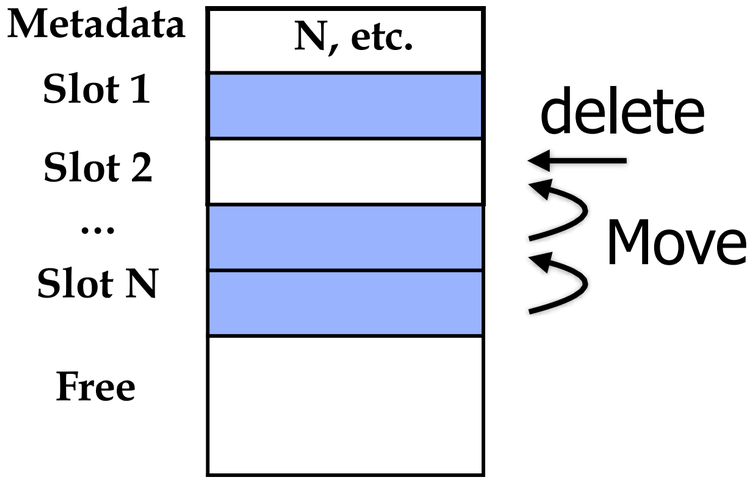


Internal Page Organization

Recall each page is 4 KB

Suppose rows are fixed-sized

How to organize rows within a slot?



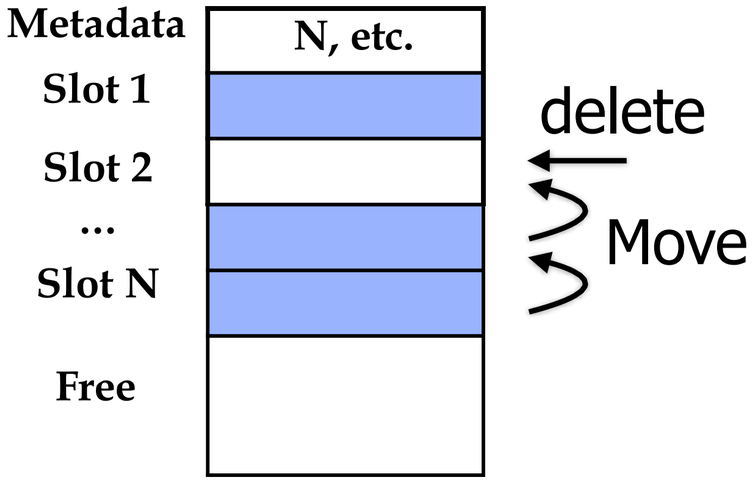
Need to reorganize due to deletes

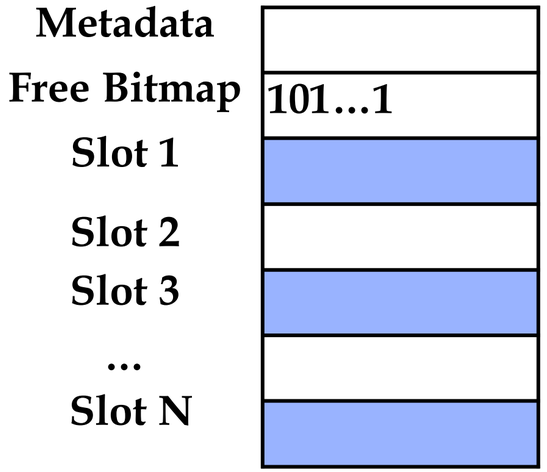
Internal Page Organization

Recall each page is 4 KB

Suppose rows are fixed-sized

How to organize rows within a slot?





Need to reorganize due to deletes

No reorganization, requires more space

Internal Page Organization

Recall each page is 4 KB

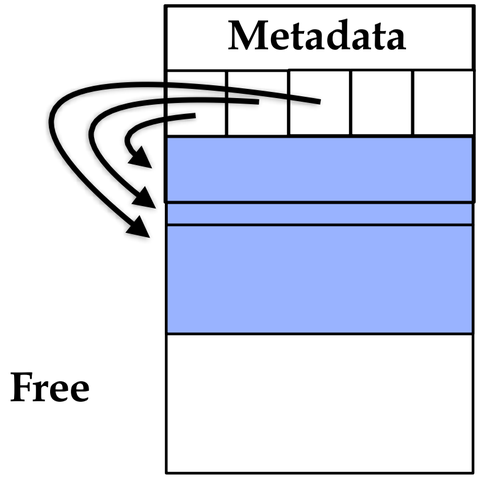
Suppose rows are **variable-length**

Solutions?

Internal Page Organization

Recall each page is 4 KB

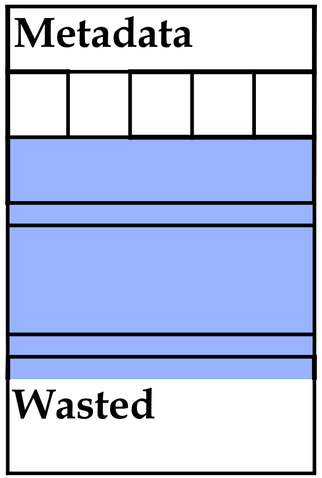
Suppose rows are **variable-length**



Internal Page Organization

Recall each page is 4 KB

Suppose rows are **variable-length**

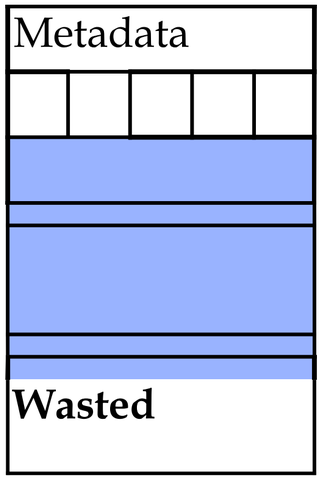


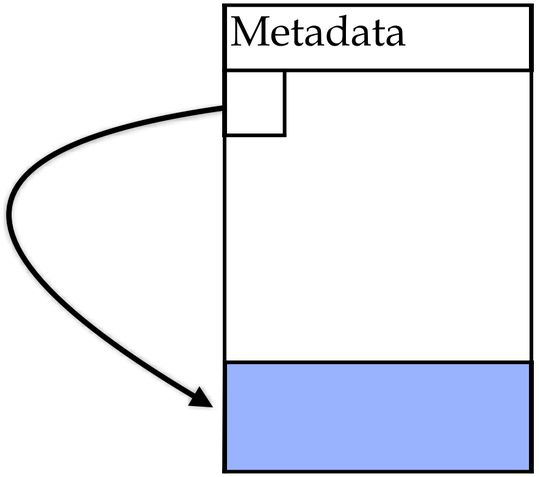
If entries are small, we waste space at the end, or we must push all content up to clear space

Internal Page Organization

Recall each page is 4 KB

Suppose rows are **variable-length**



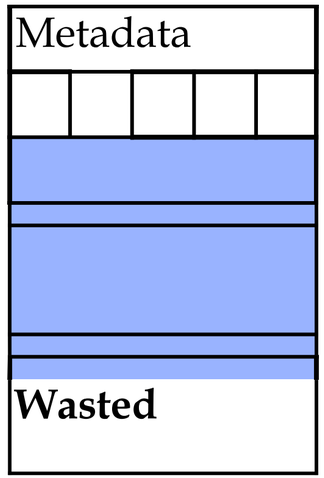


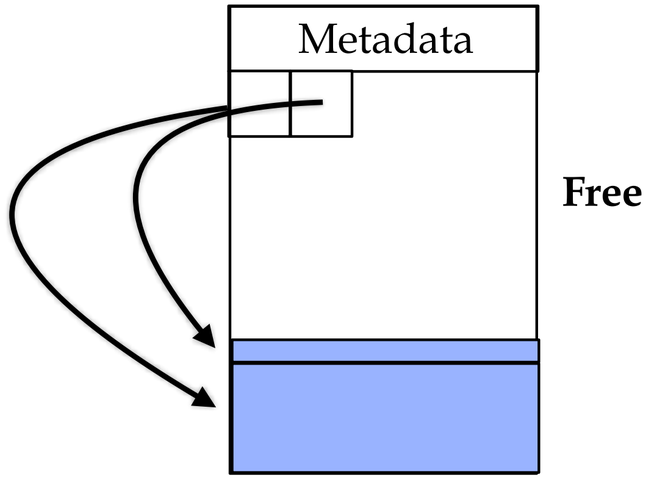
Store data from end of page

Internal Page Organization

Recall each page is 4 KB

Suppose rows are **variable-length**

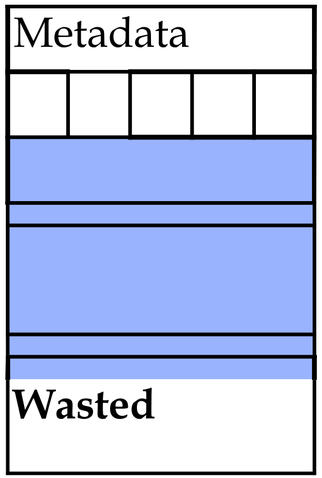


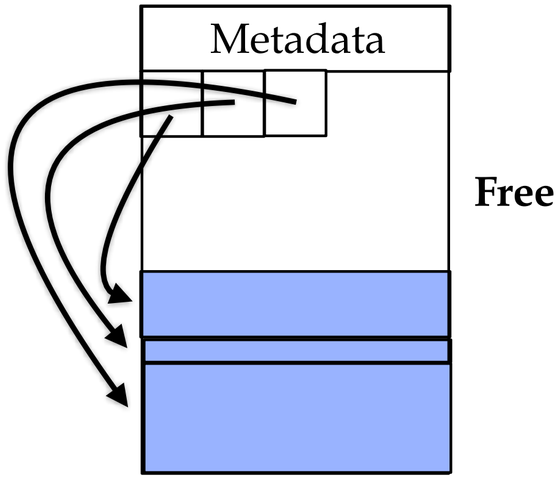


Internal Page Organization

Recall each page is 4 KB

Suppose rows are **variable-length**

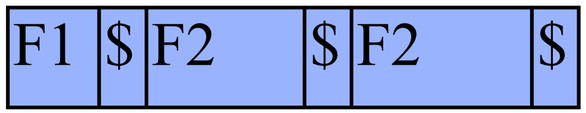




Minimal space wastage,
and no need to move data

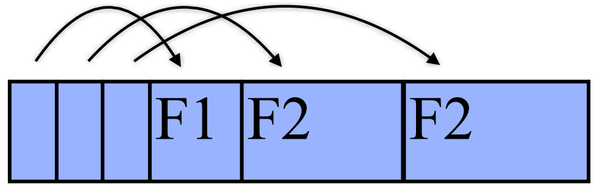
Variable-Sized Record Organization

Delimiters



Smaller
No random access

Pointers

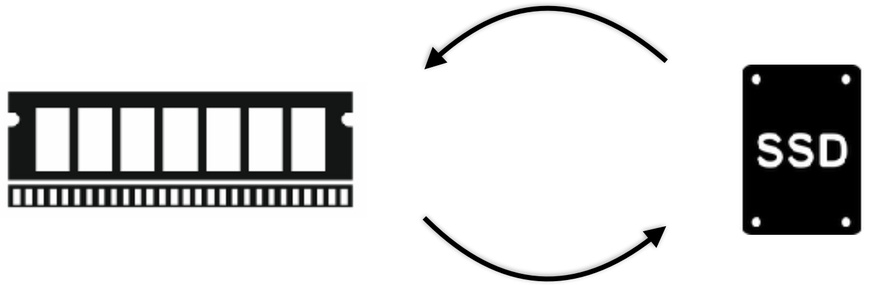


More space
Random access (faster)

Break

Then let’s now move to buffer management

Buffer Management

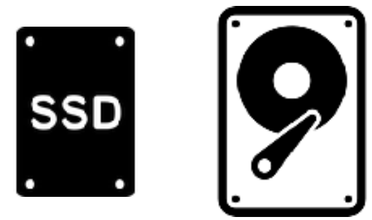


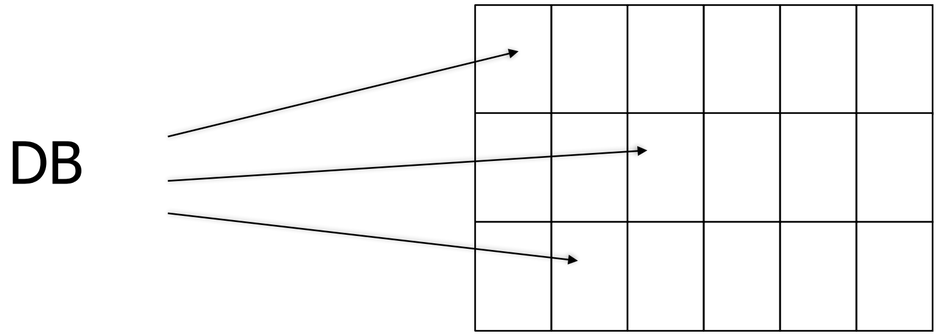
Database System Technology

Niv Dayan

Context

A DB is reading and writing aligned 4KB storage pages



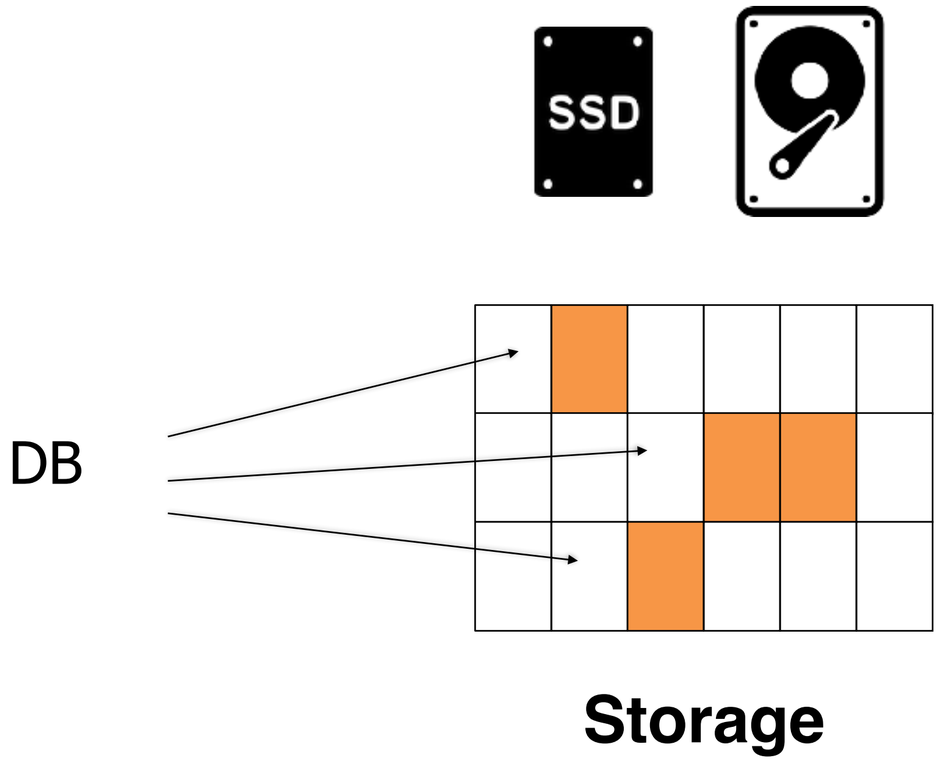


Storage

Context

A DB is reading and writing aligned 4KB storage pages

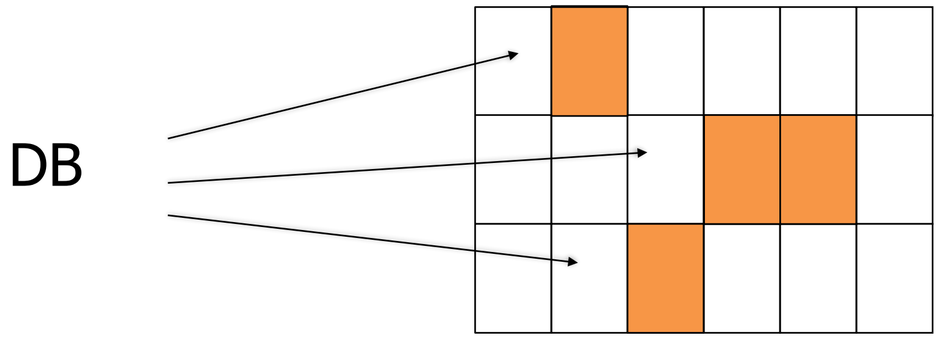
Suppose orange pages are frequently accessed (“hot”)



Context

A DB is reading and writing aligned 4KB storage pages

Suppose orange pages are frequently accessed ("hot")

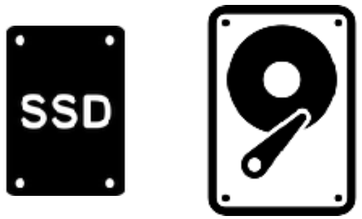


Retrieving these pages over and over is expensive!

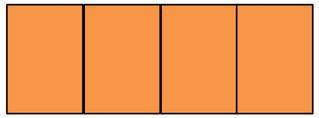
Storage

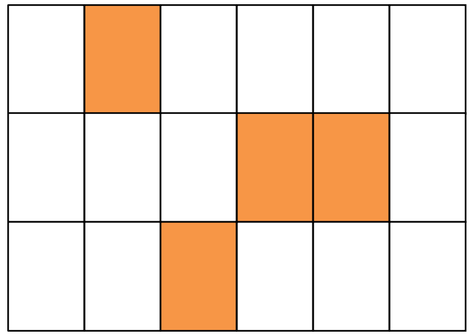
Buffer Pool

Keep copies of hot pages in memory



DB



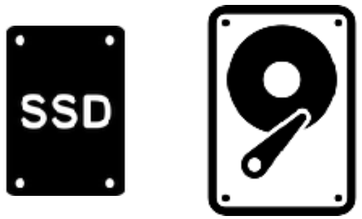


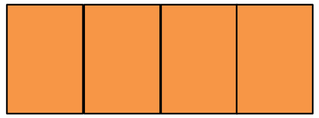
memory

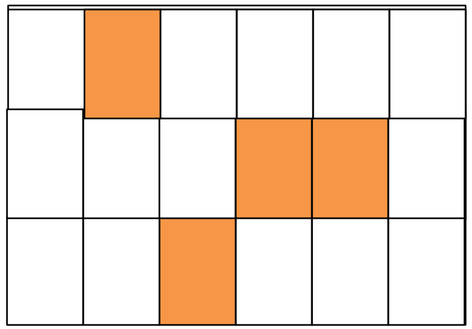
Storage

Buffer Pool

How to structure this buffer pool?







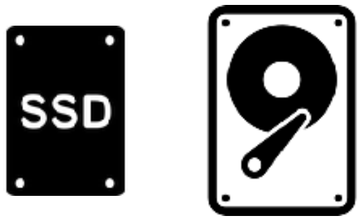
memory

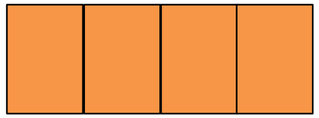
Storage

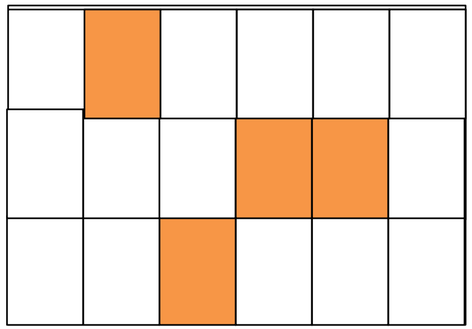
Buffer Pool

How to structure this buffer pool?

hash table (more details later)





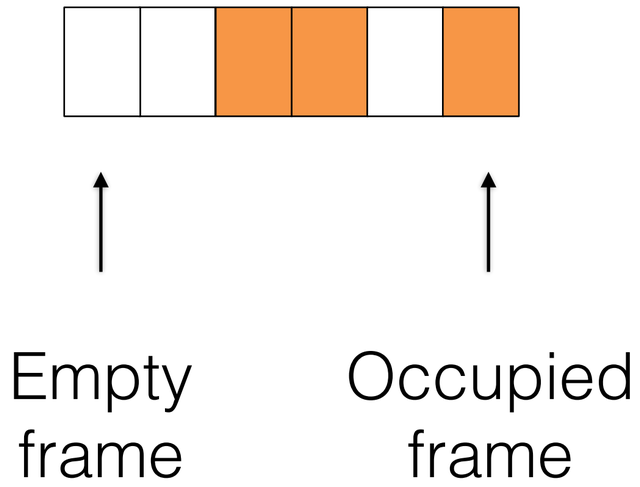


memory

Storage

Buffer Pool

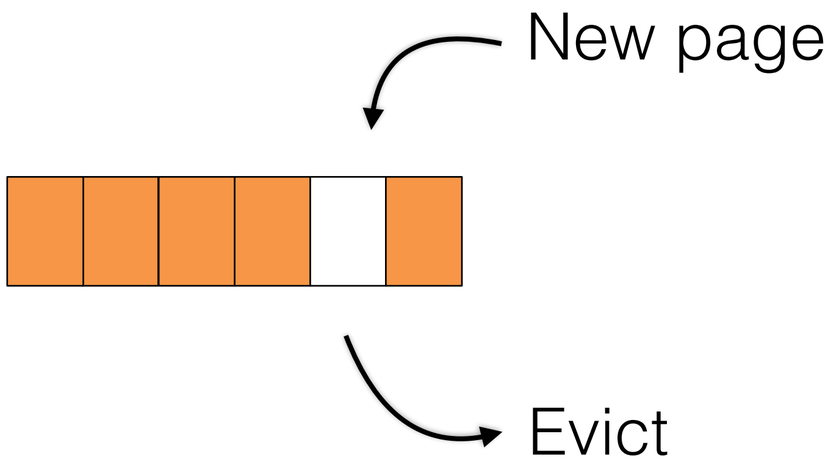
Consist of frames, each containing one page of data (e.g., 4 KB)



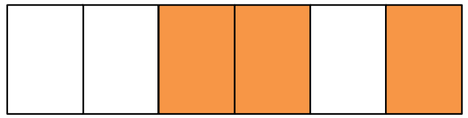
Buffer Pool

Consist of frames, each containing one page of data (e.g., 4 KB)

Eventually it fills up. Must evict pages to clear space.



Buffer Pool

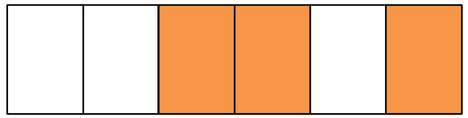


Each frame must keep some metadata

(1) ?

(2) ?

Buffer Pool



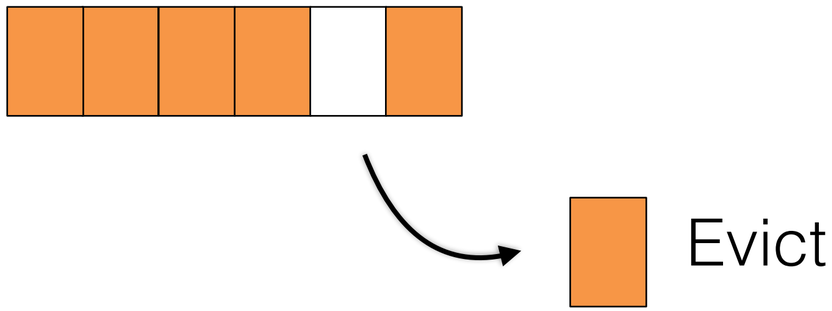
Each frame must keep some metadata

(1) **Pin count** - How many users are currently using this page

(2) **Dirty flag** - indicates whether the page has been updated

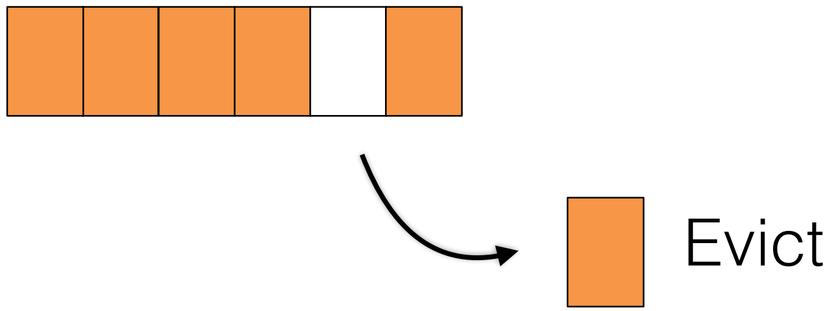
Eviction Policy

Which page to evict when we run out of space?



Eviction Policy

Which page to evict when we run out of space?



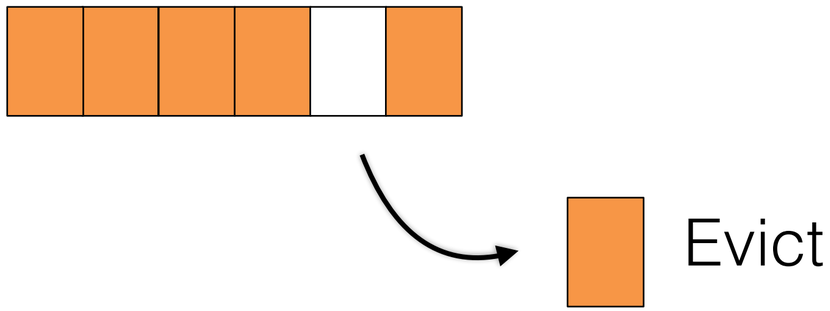
Considerations:

(1) Avoid evicting a page that is likely to be used again

(2) Avoid excessive metadata or CPU overheads to make decision

Eviction Policy

Which page to evict when we run out of space?



Big impact on number of I/Os and CPU efficiency

Depends on the access pattern

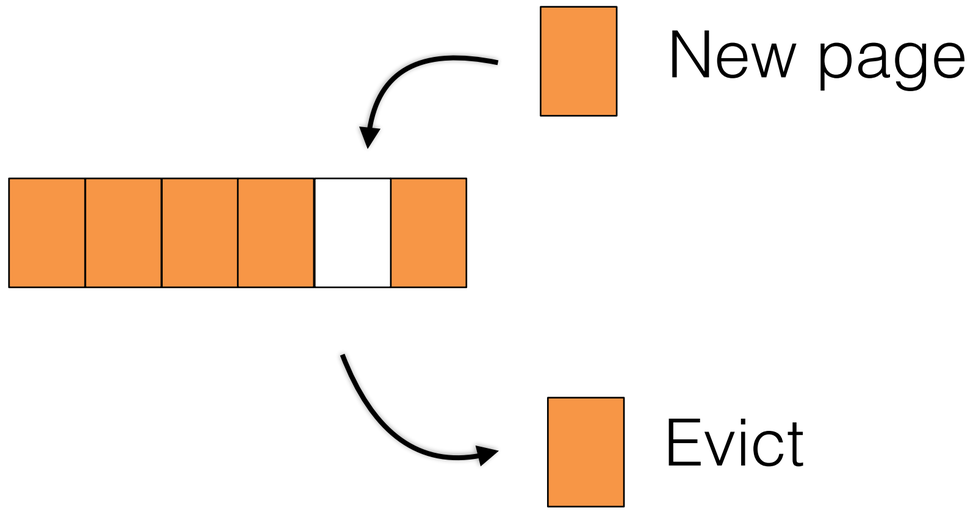
We’ll cover 5 eviction policies

Random Eviction

Evict whichever page collides in the hash table with a new page

Pros: ?

Con: ?

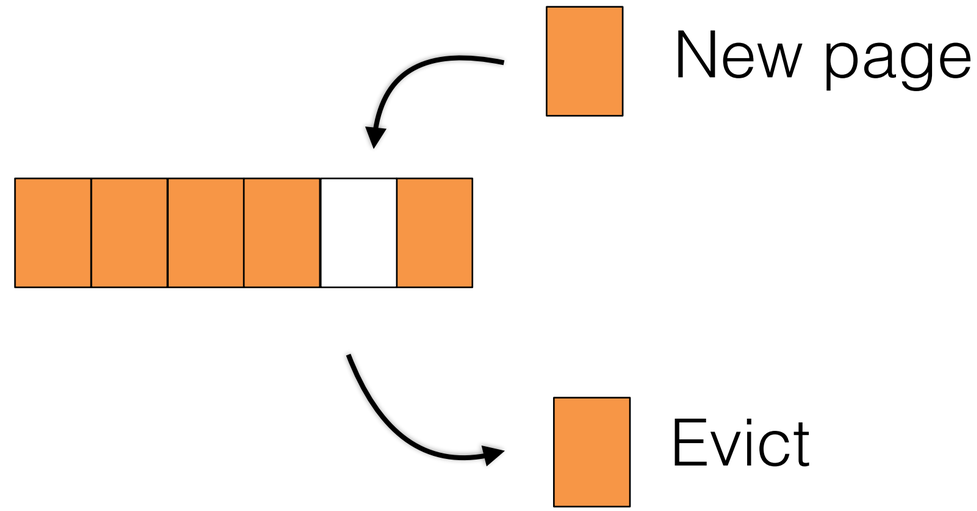


Random Eviction

Evict whichever page collides in the hash table with a new page

Pros: **Simple, CPU-efficient, no extra metadata**

Con: ?

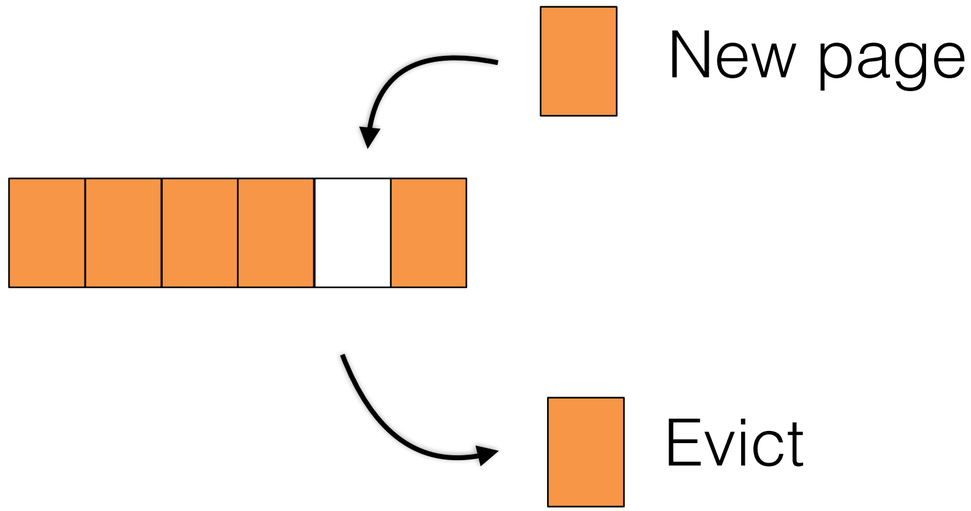


Random Eviction

Evict whichever page collides in the hash table with a new page

Pros: Simple, CPU-efficient, no extra metadata

Con: **May evict a frequently used page**



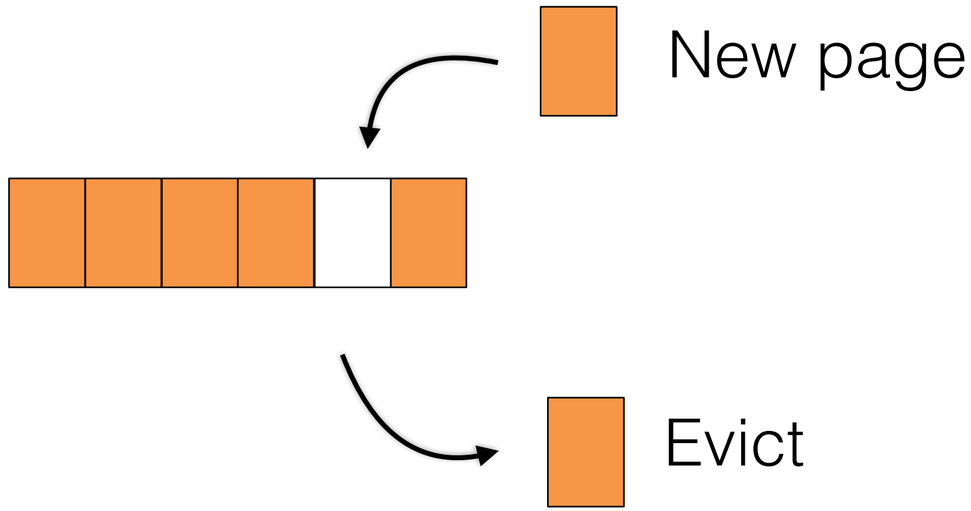
Random Eviction

Evict whichever page collides in the hash table with a new page

Pros: Simple, CPU-efficient, no extra metadata

Con: **May evict a frequently used page**

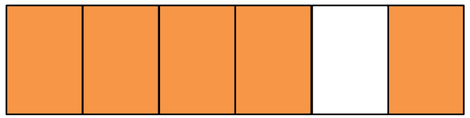
Can we improve this?



First in First Out (FIFO)

Evict Page that was inserted the longest time ago

Rationale?

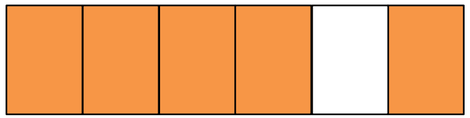


First in First Out (FIFO)

Evict Page that was inserted the longest time ago

Rationale? Less likely to be used again

Implementation?



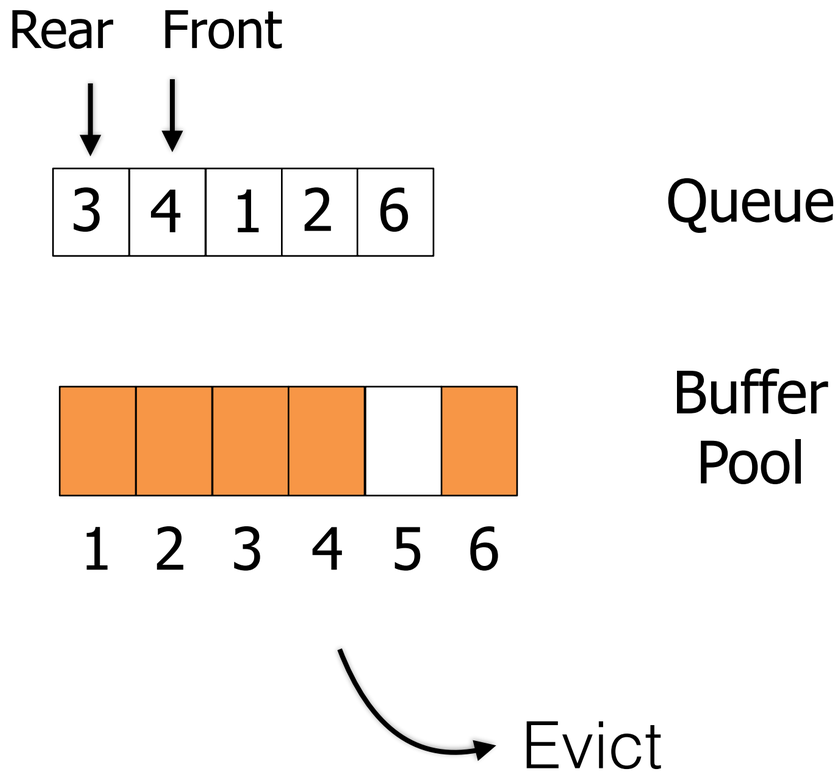
First in First Out (FIFO)

Evict Page that was inserted the longest time ago

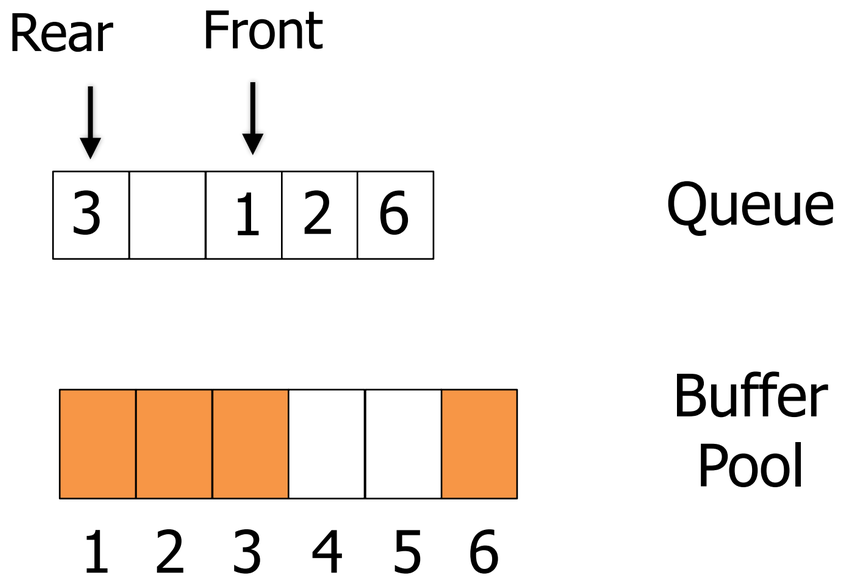
Rationale? Less likely to be used again

Implementation? **Using a queue**

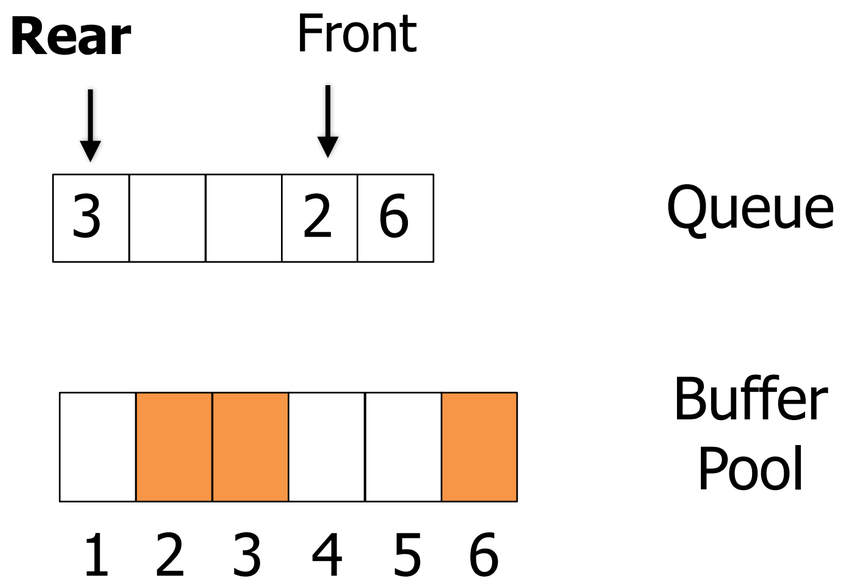
(i.e., array with front/rear pointers)



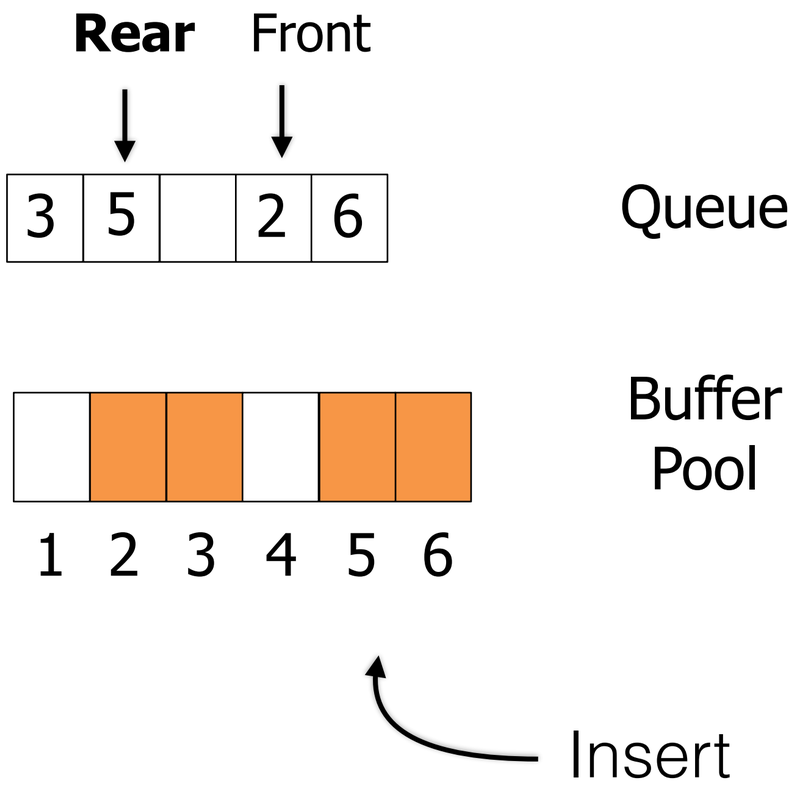
First in First Out (FIFO)



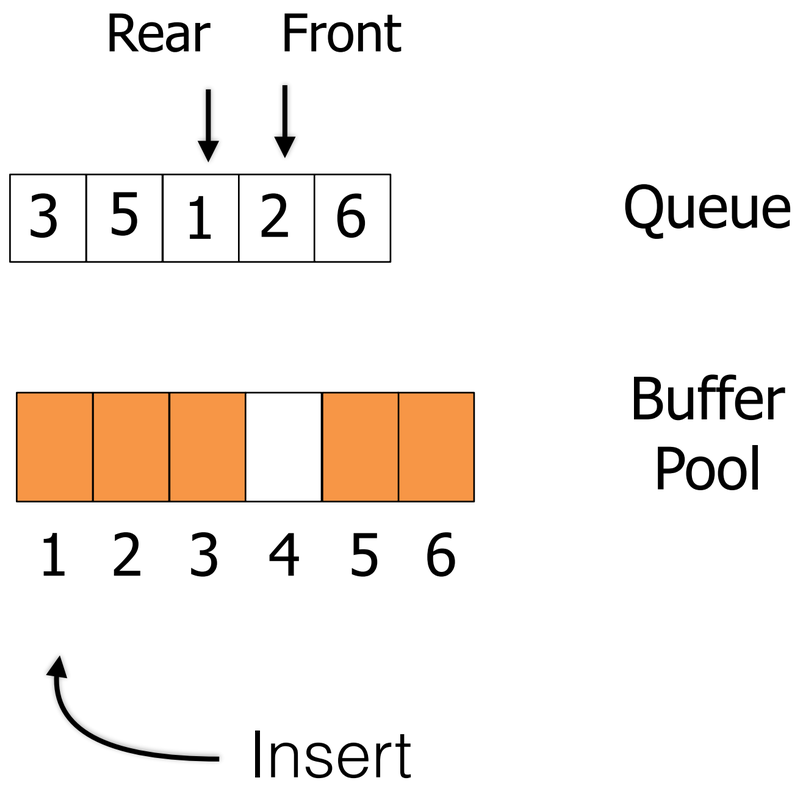
First in First Out (FIFO)



First in First Out (FIFO)

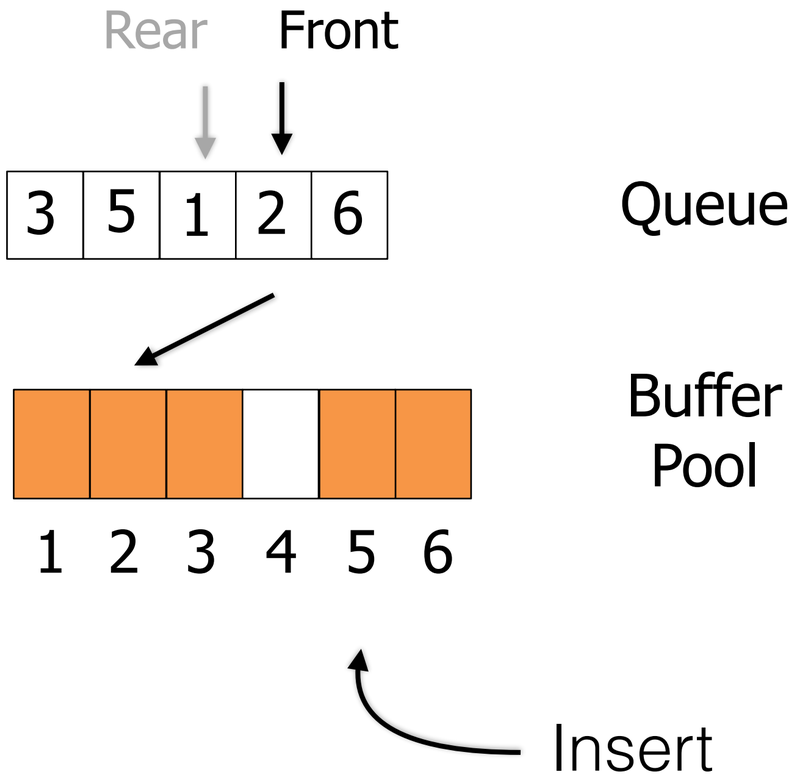


First in First Out (FIFO)



First in First Out (FIFO)

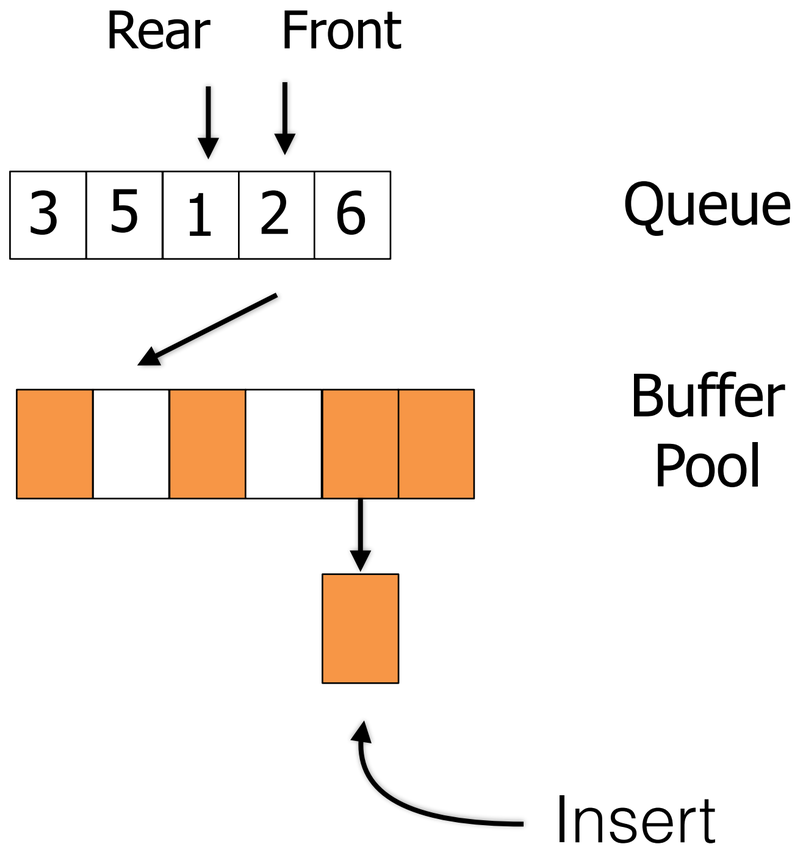
Unlike the random policy, pages we evict now have different frames than pages we insert.



First in First Out (FIFO)

Unlike the random policy, pages we evict now have different frames than pages we insert.

We need a hash collision resolution algo. (e.g., linear probing, chaining)

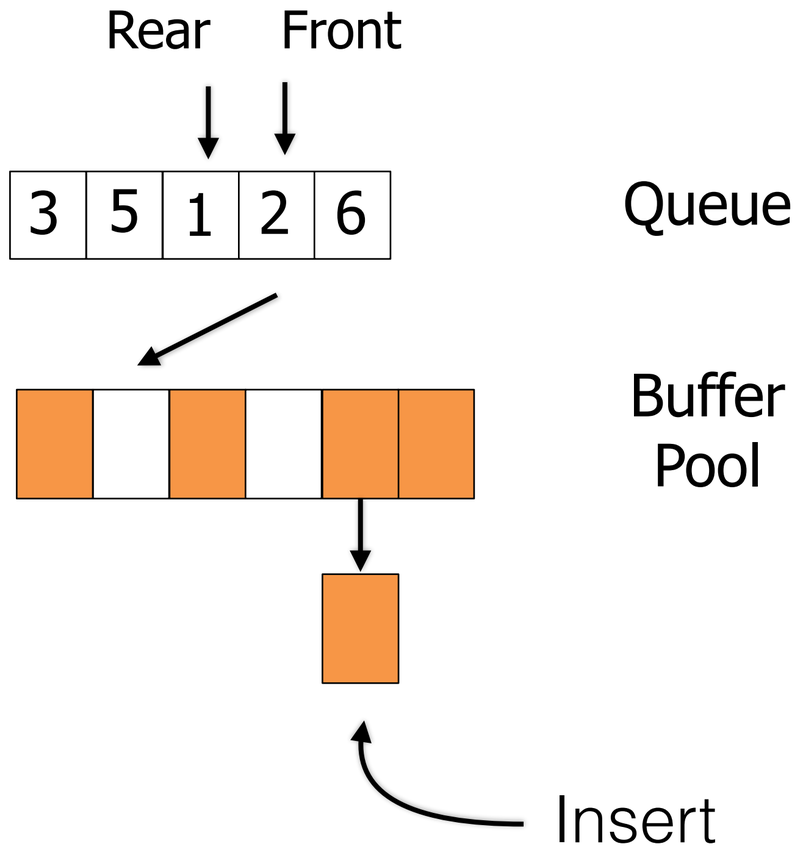


First in First Out (FIFO)

Unlike the random policy, pages we evict now have different frames than pages we insert.

We need a hash collision resolution algo. (e.g., linear probing, chaining)

Need 10-20% extra capacity in the table to reduce likelihood of collisions



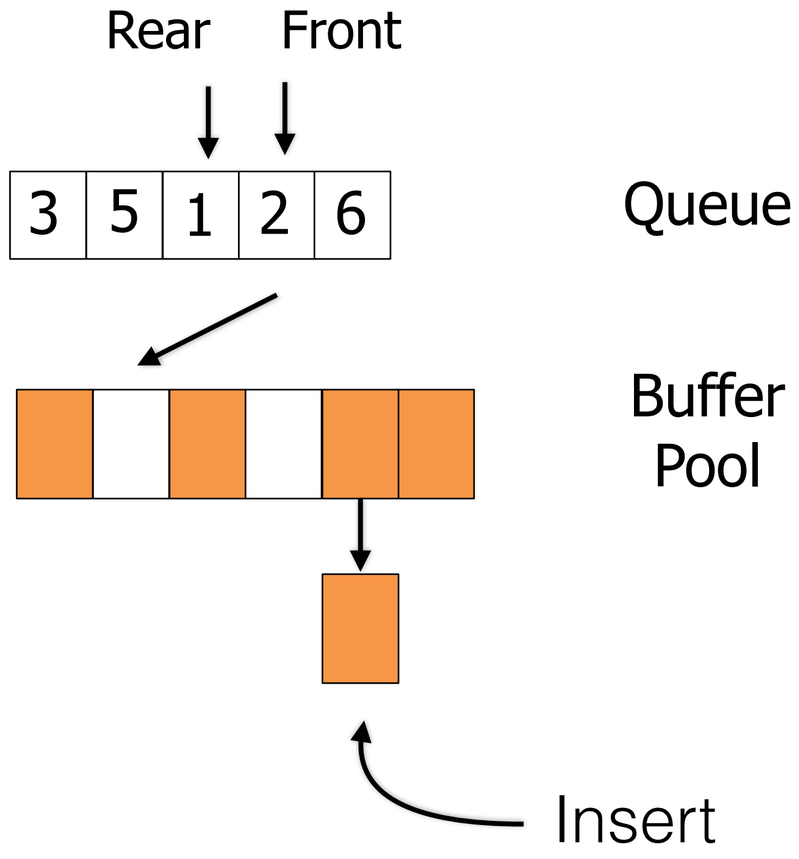
First in First Out (FIFO)

Unlike the random policy, pages we evict now have different frames than pages we insert.

We need a hash collision resolution algo. (e.g., linear probing, chaining)

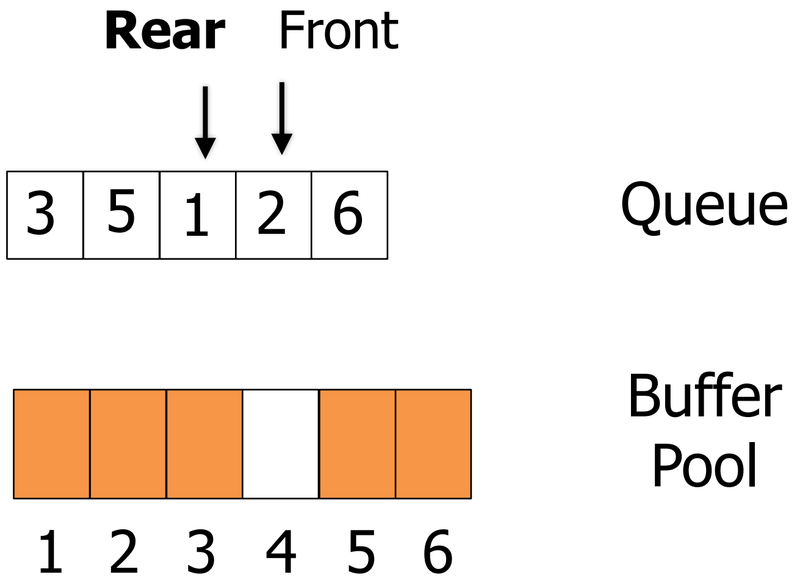
Need 10-20% extra capacity in the table to reduce likelihood of collisions

Extra space and CPU cost, but necessary for all policies but the random one



First in First Out (FIFO)

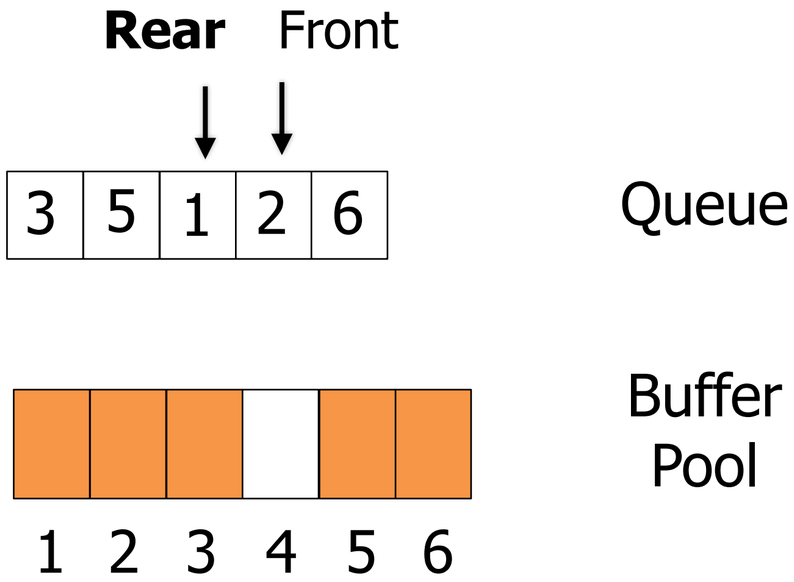
Problem?



First in First Out (FIFO)

Problem?

Oldest page may still be frequently used.

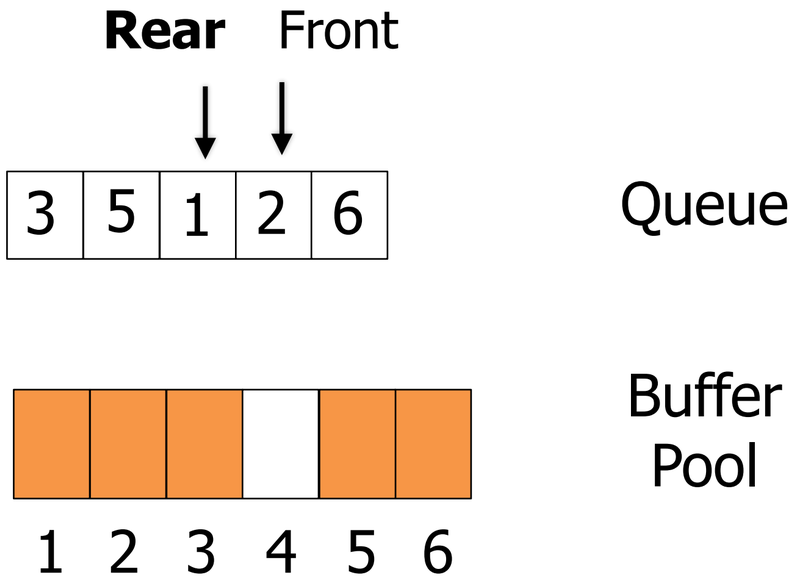


First in First Out (FIFO)

Problem?

Oldest page may still be frequently used.

Can we address this?



Least Recently Used (LRU)

Evict page that was used last the longest time ago

Least Recently Used (LRU)

Evict page that was used last the longest time ago

Implementation?

Least Recently Used (LRU)

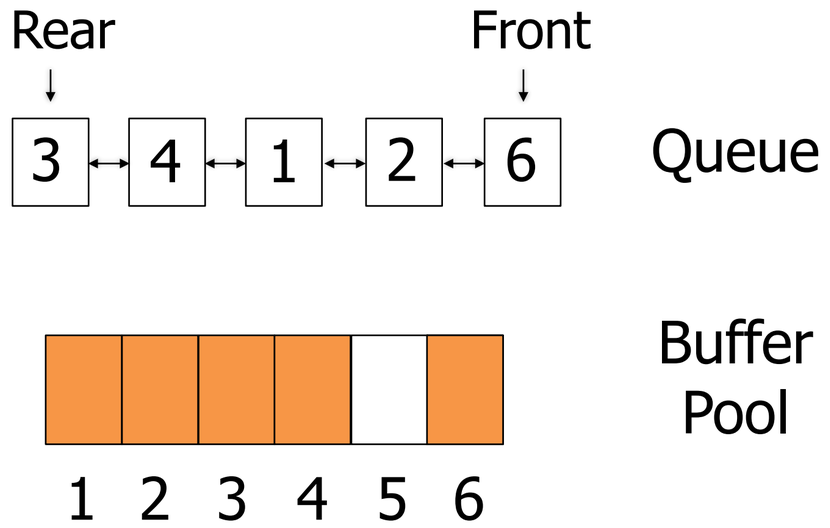
Evict page that was used last the longest time ago

Implementation? **A random-access queue**

i.e., We must be able to move an element at a random location back to the front

Least Recently Used (LRU)

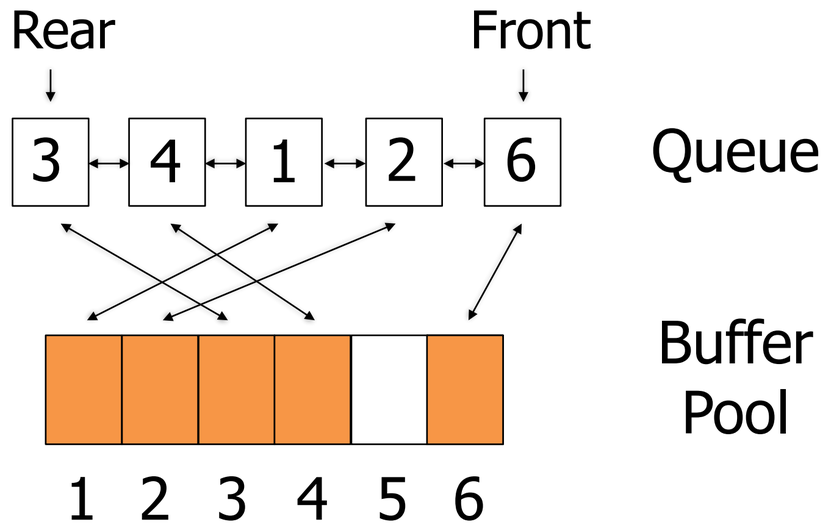
Use a doubly-linked list as the queue



Least Recently Used (LRU)

Use a doubly-linked list as the queue

With pointers linking nodes and buckets

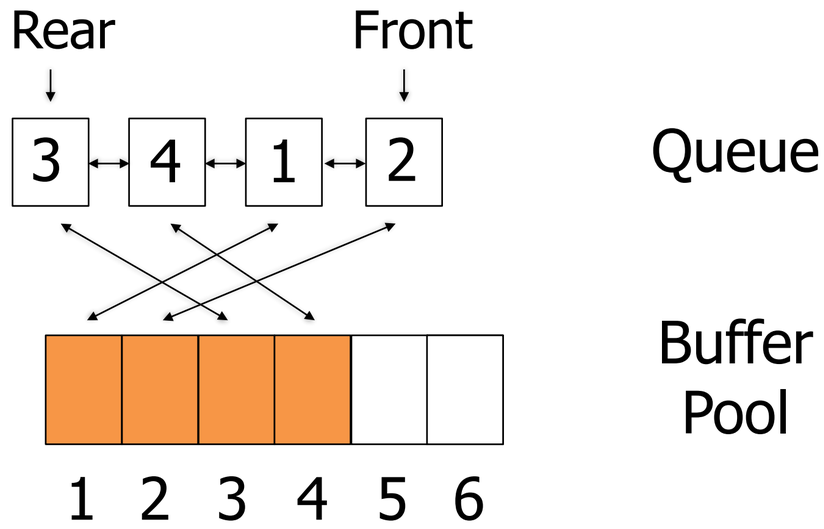


Least Recently Used (LRU)

Use a doubly-linked list as the queue

With pointers linking nodes and buckets

Load to rear and evict front (as before)

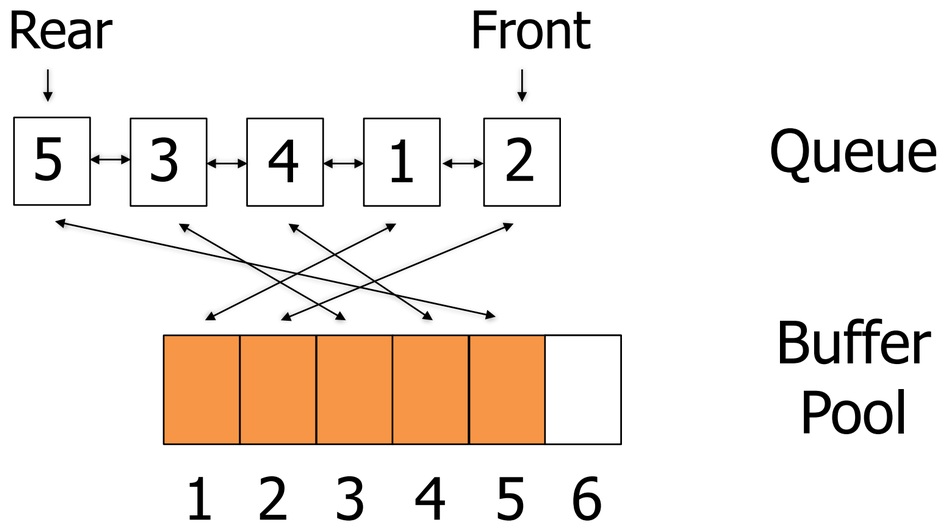


Least Recently Used (LRU)

Use a doubly-linked list as the queue

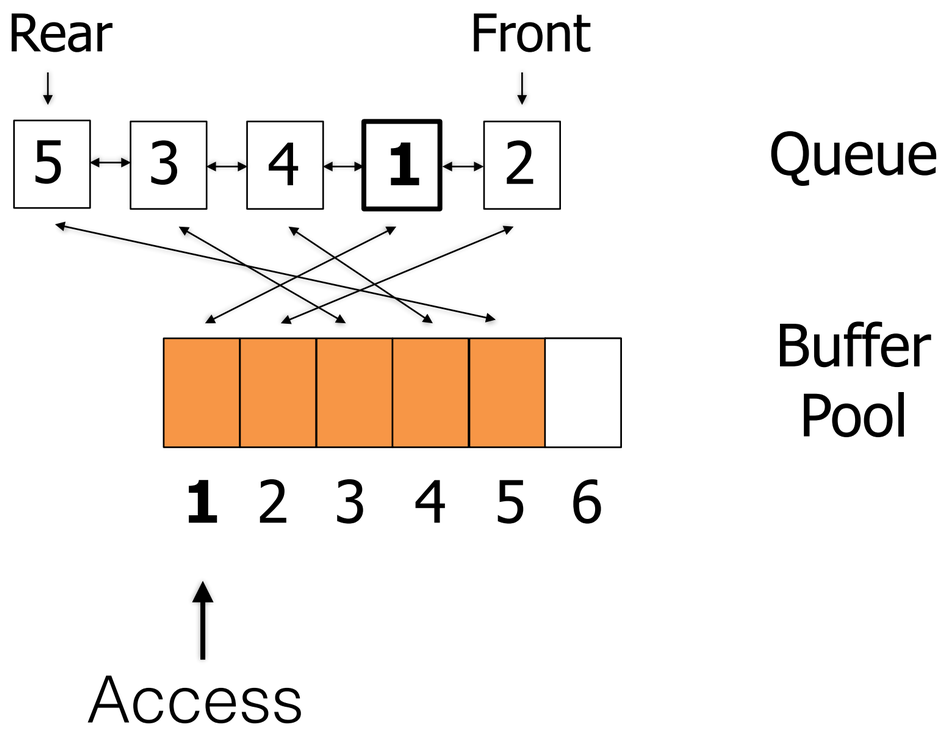
With pointers linking nodes and buckets

Load to rear and evict front (as before)



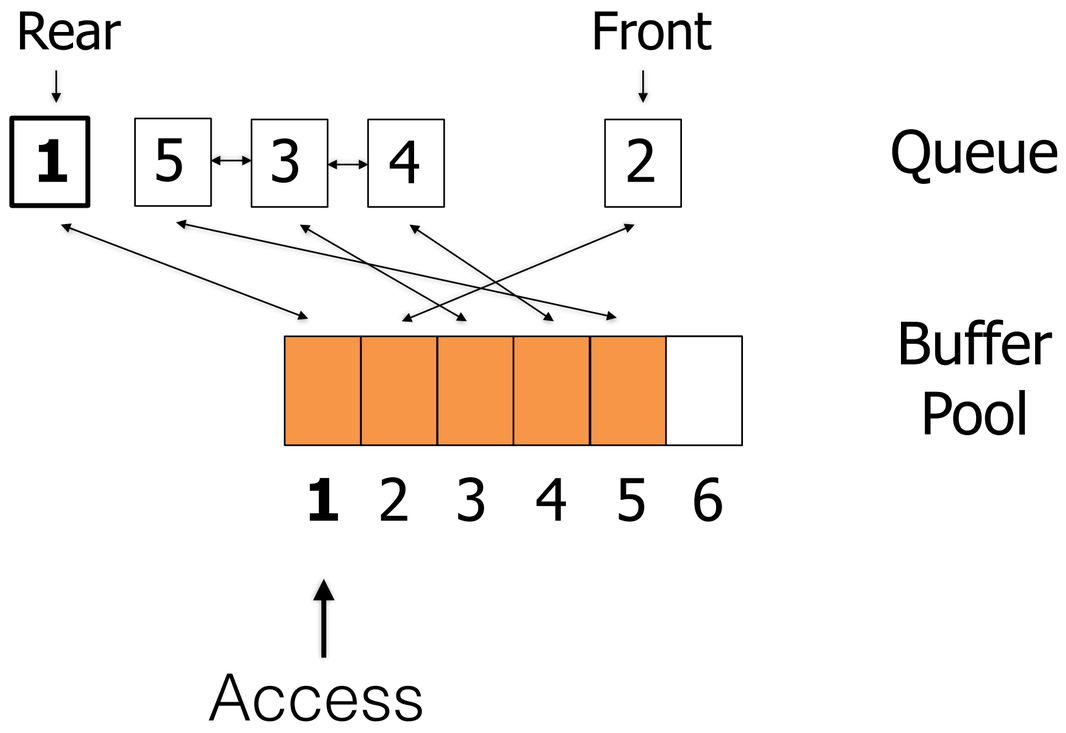
Least Recently Used (LRU)

During access, return entry to rear



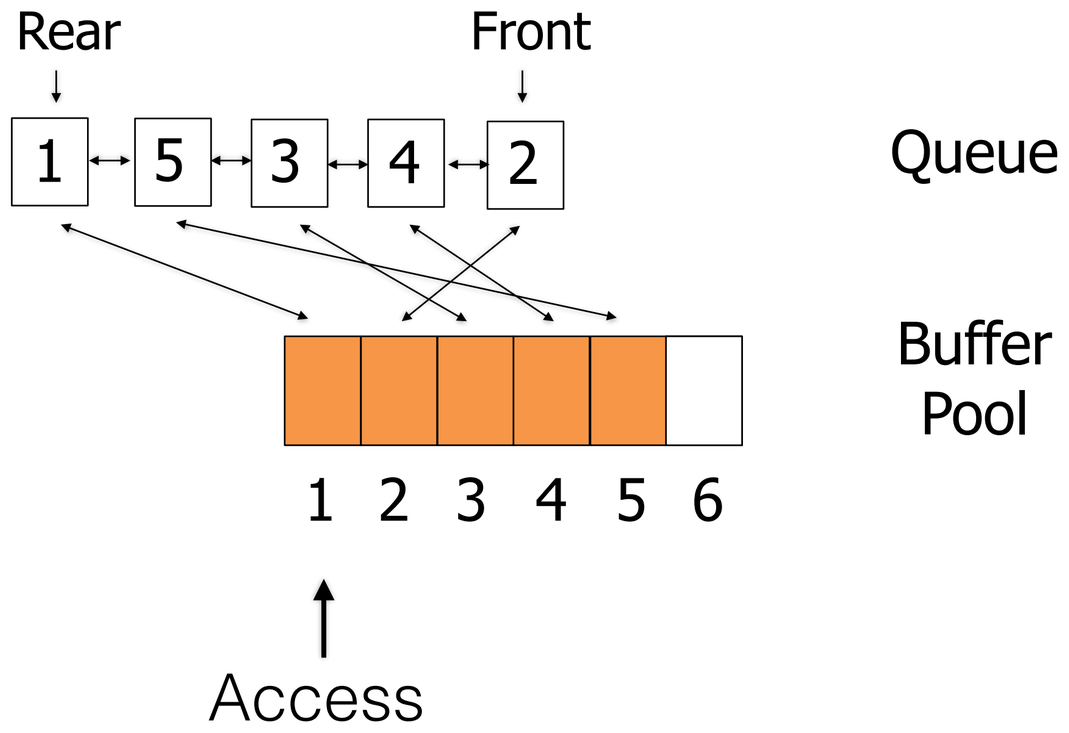
Least Recently Used (LRU)

Implementation? Doubly-linked list



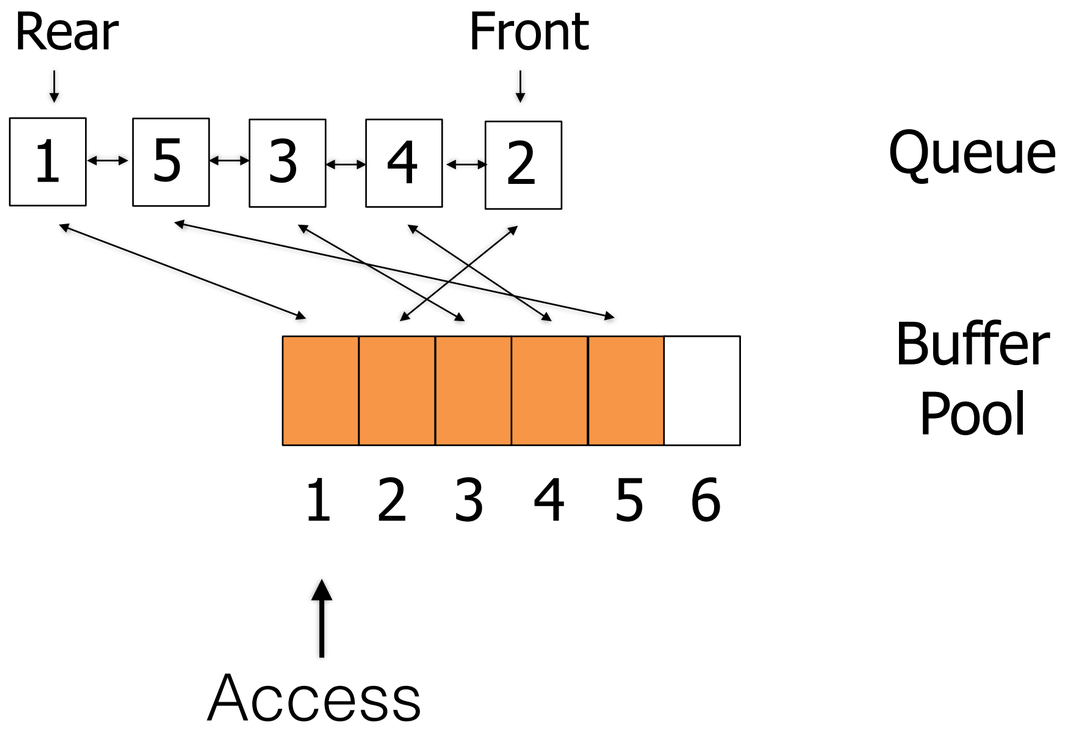
Least Recently Used (LRU)

Implementation? Doubly-linked list



Least Recently Used (LRU)

Problems?



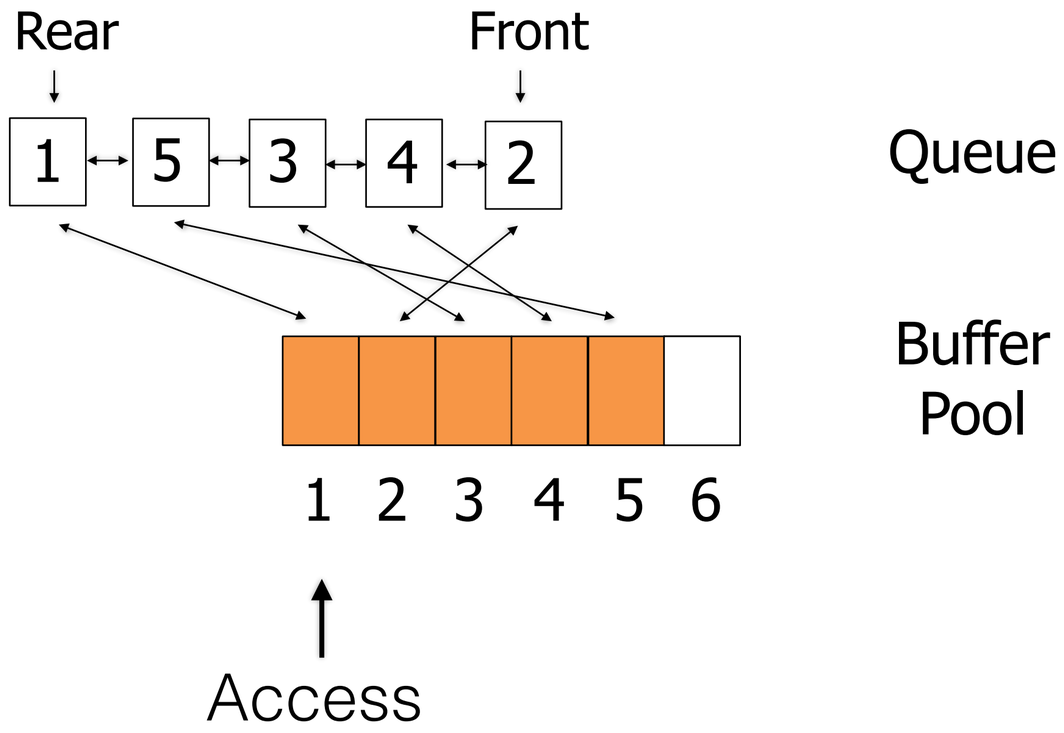
Least Recently Used (LRU)

Problems?

(1) CPU overhead to update queue for each access

(2) Linked lists are less efficient than arrays due to pointer chasing

(3) Metadata overhead for pointers



Least Recently Used (LRU)

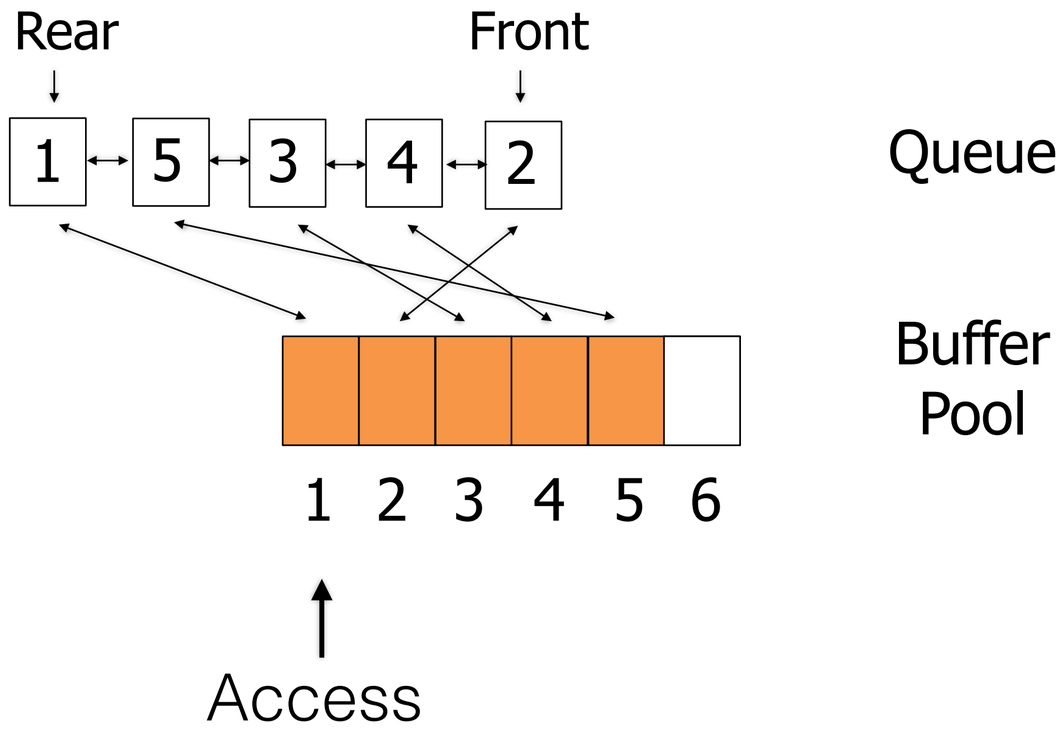
Problems?

(1) CPU overhead to update queue for each access

(2) Linked lists are less efficient than arrays due to pointer chasing

(3) Metadata overhead for pointers

Better ideas? :)



Clock

Traverse hash table circularly as a clock.
Evict any entry not used since last traversal.

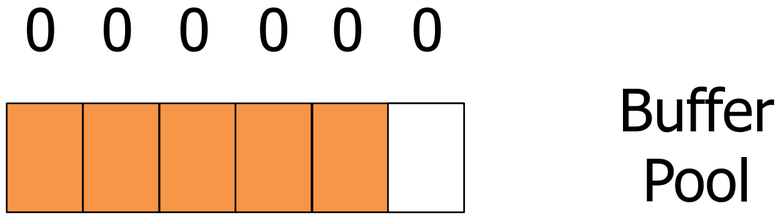
Implementation?

Buffer Pool

Clock

Traverse hash table circularly as a clock.
Evict any entry not used since last traversal.

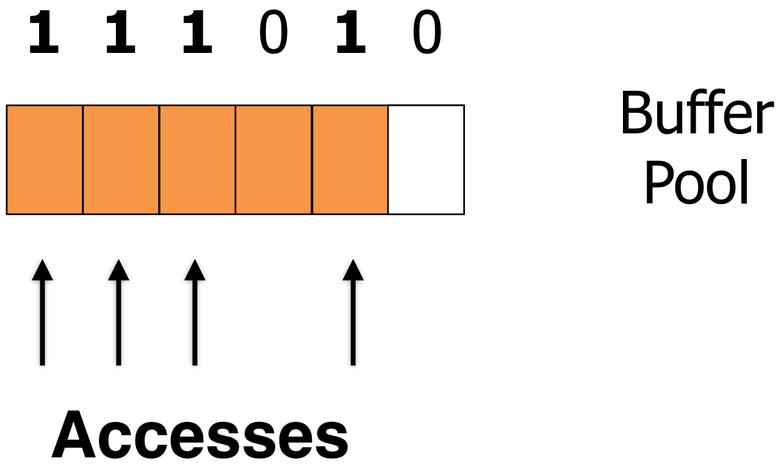
Implementation? Bitmap



Clock

Traverse hash table circularly as a clock.
Evict any entry not used since last traversal.

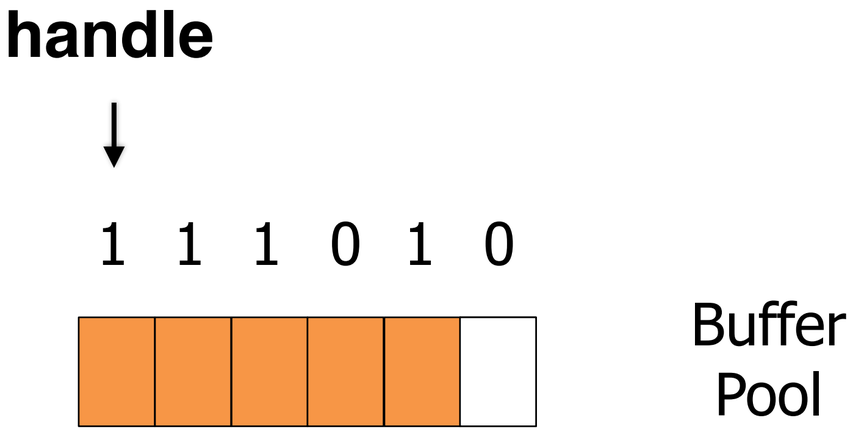
Implementation? Bitmap



Clock

Traverse hash table circularly as a clock.
Evict any entry not used since last traversal.

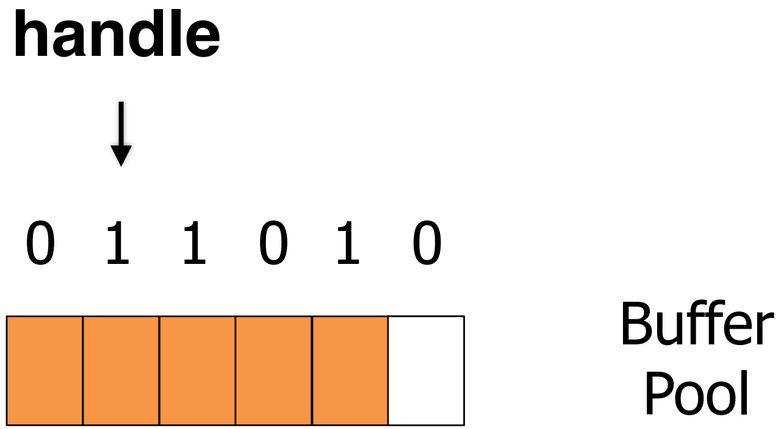
Implementation? Bitmap



Clock

Traverse hash table circularly as a clock.
Evict any entry not used since last traversal.

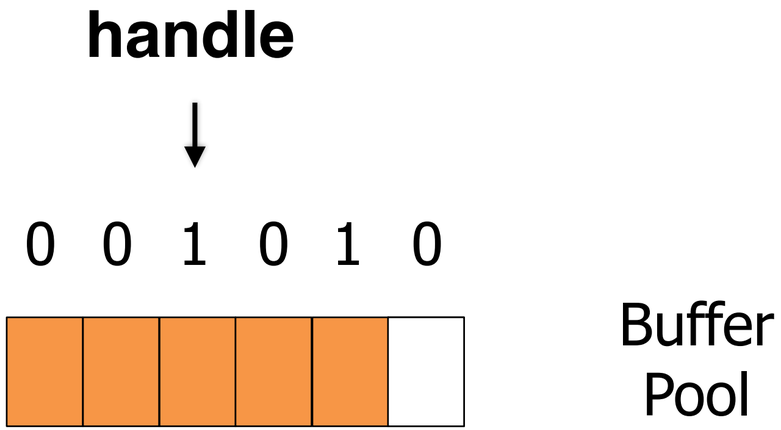
Implementation? Bitmap



Clock

Traverse hash table circularly as a clock.
Evict any entry not used since last traversal.

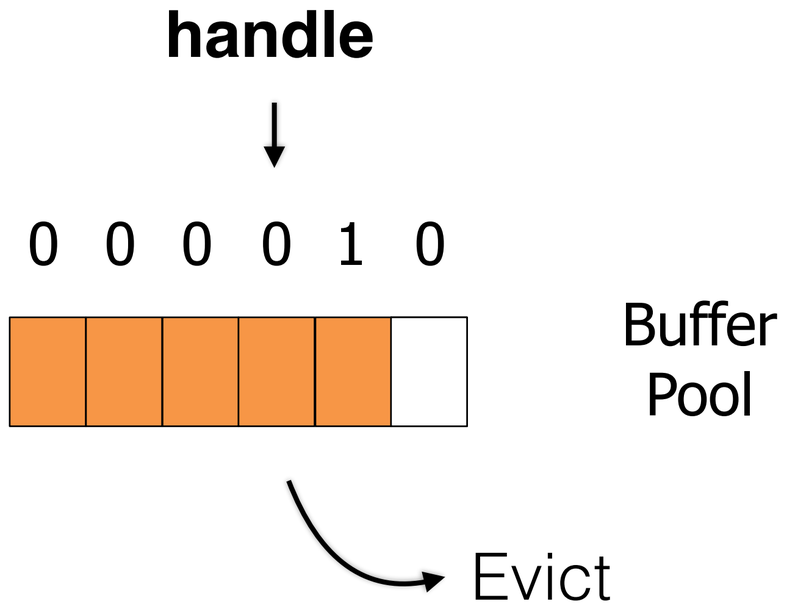
Implementation? Bitmap



Clock

Traverse hash table circularly as a clock.
Evict any entry not used since last traversal.

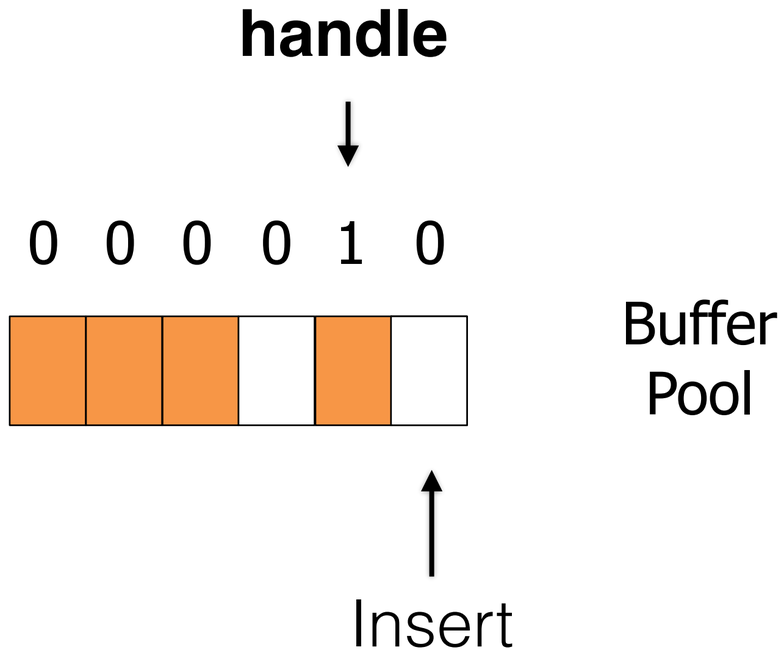
Implementation? Bitmap



Clock

Traverse hash table circularly as a clock.
Evict any entry not used since last traversal.

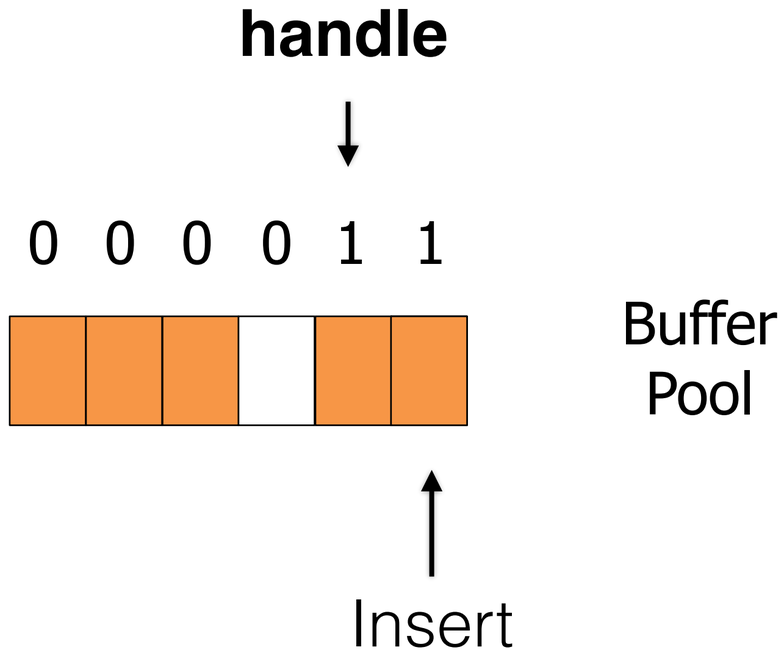
Implementation? Bitmap



Clock

Traverse hash table circularly as a clock.
Evict any entry not used since last traversal.

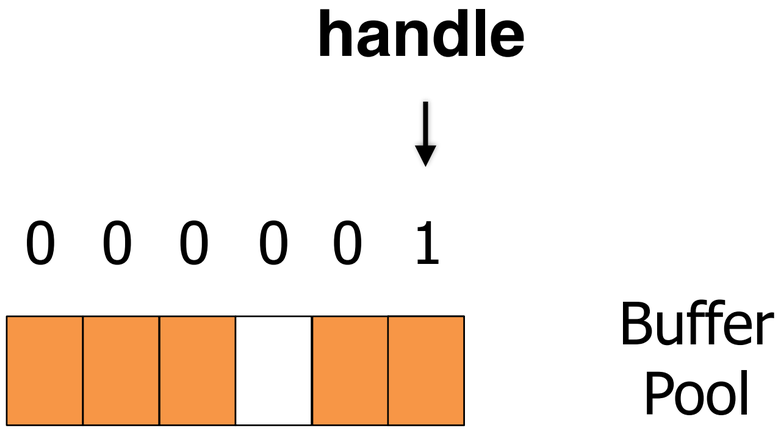
Implementation? Bitmap



Clock

Traverse hash table circularly as a clock.
Evict any entry not used since last traversal.

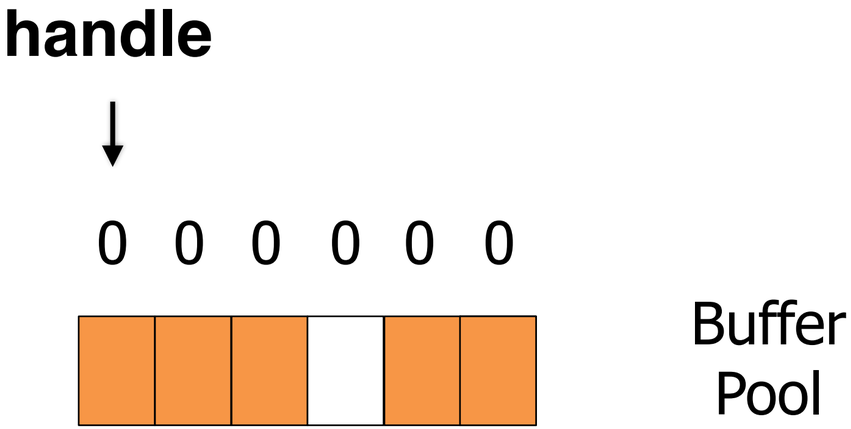
Implementation? Bitmap



Clock

Traverse hash table circularly as a clock.
Evict any entry not used since last traversal.

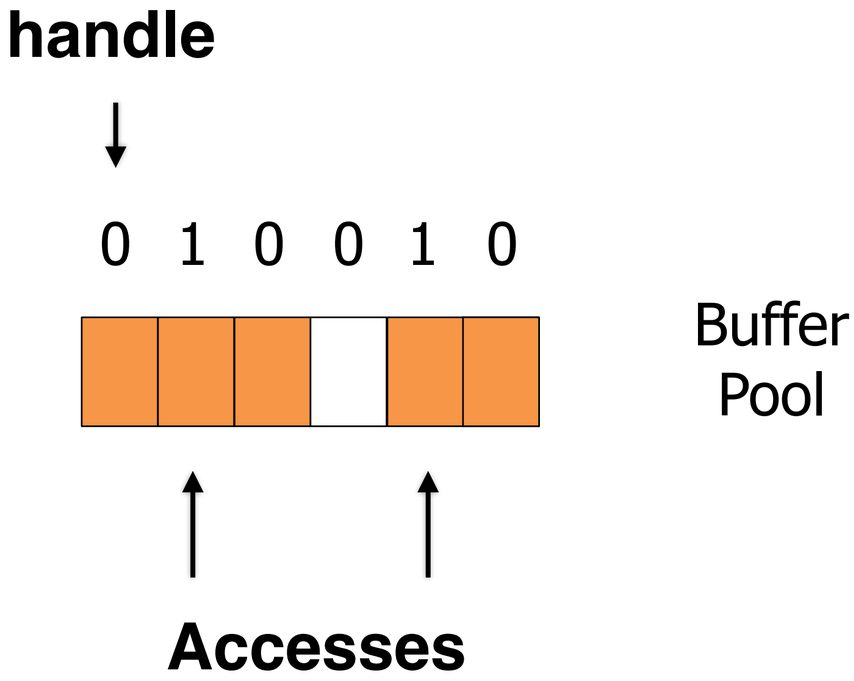
Implementation? Bitmap



Clock

Traverse hash table circularly as a clock.
Evict any entry not used since last traversal.

Implementation? Bitmap



Clock

Traverse hash table circularly as a clock.
Evict any entry not used since last traversal.

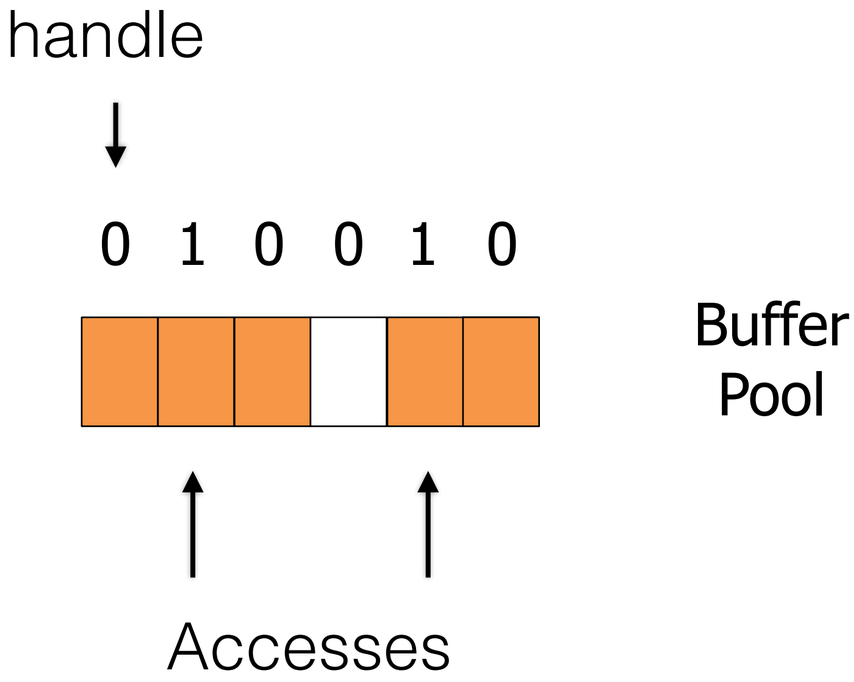
Advantages

(1) lower overheads as there is no queue

(2) bitmap takes little extra space

Disadvantages

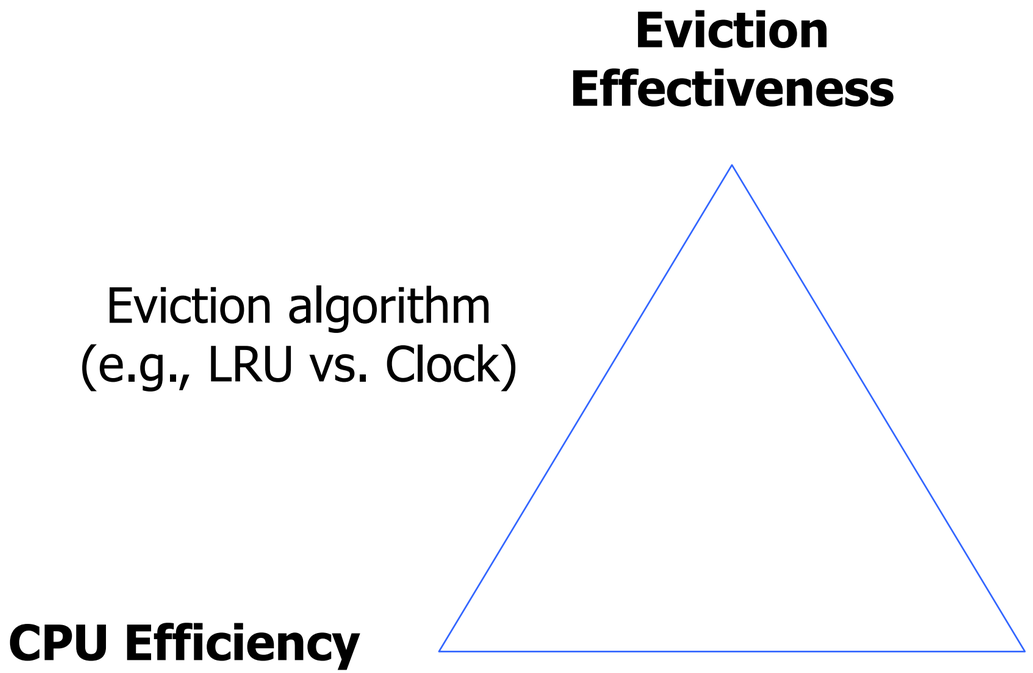
(1) can evict “hotter” pages than LRU,
But still better than FIFO



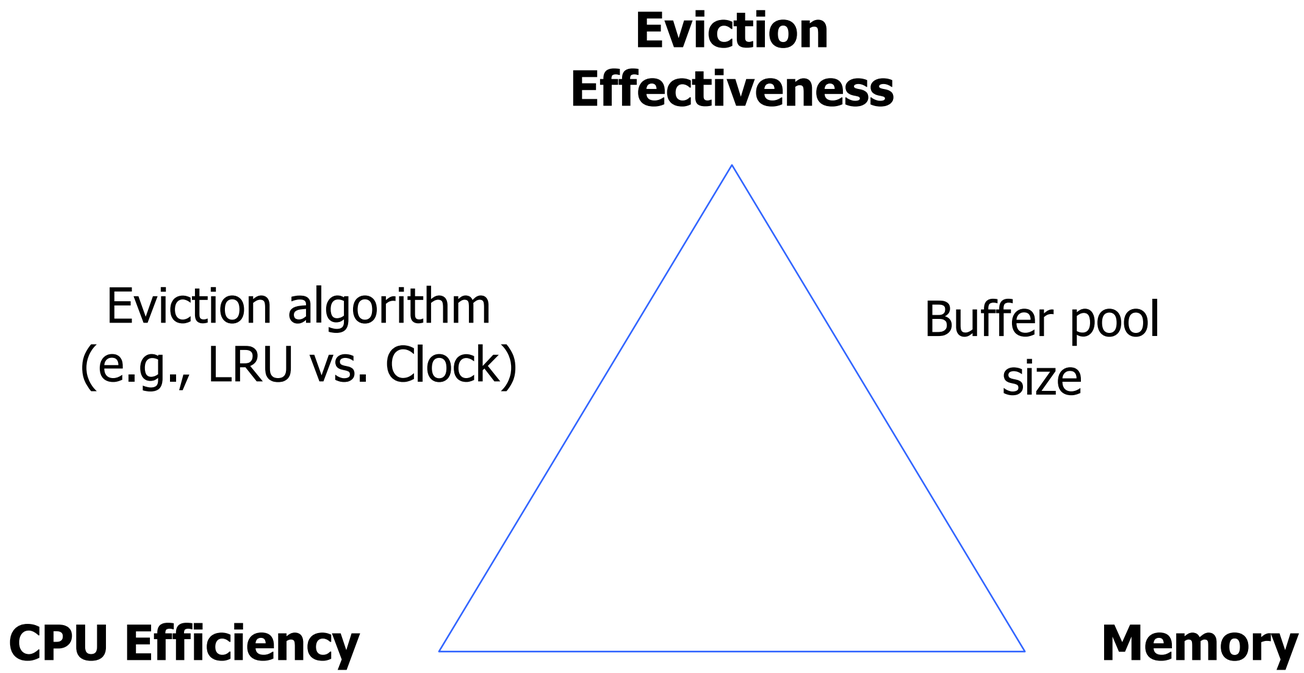
Summary

	Eviction Effectiveness	CPU
Random	Worst	Best
FIFO	Moderate	Good
LRU	Best	Worst
Clock	Close to Best	Good

Two trade-offs



Two trade-offs



LRU and Clock are good for small random reads. How do they respond to large sequential reads?