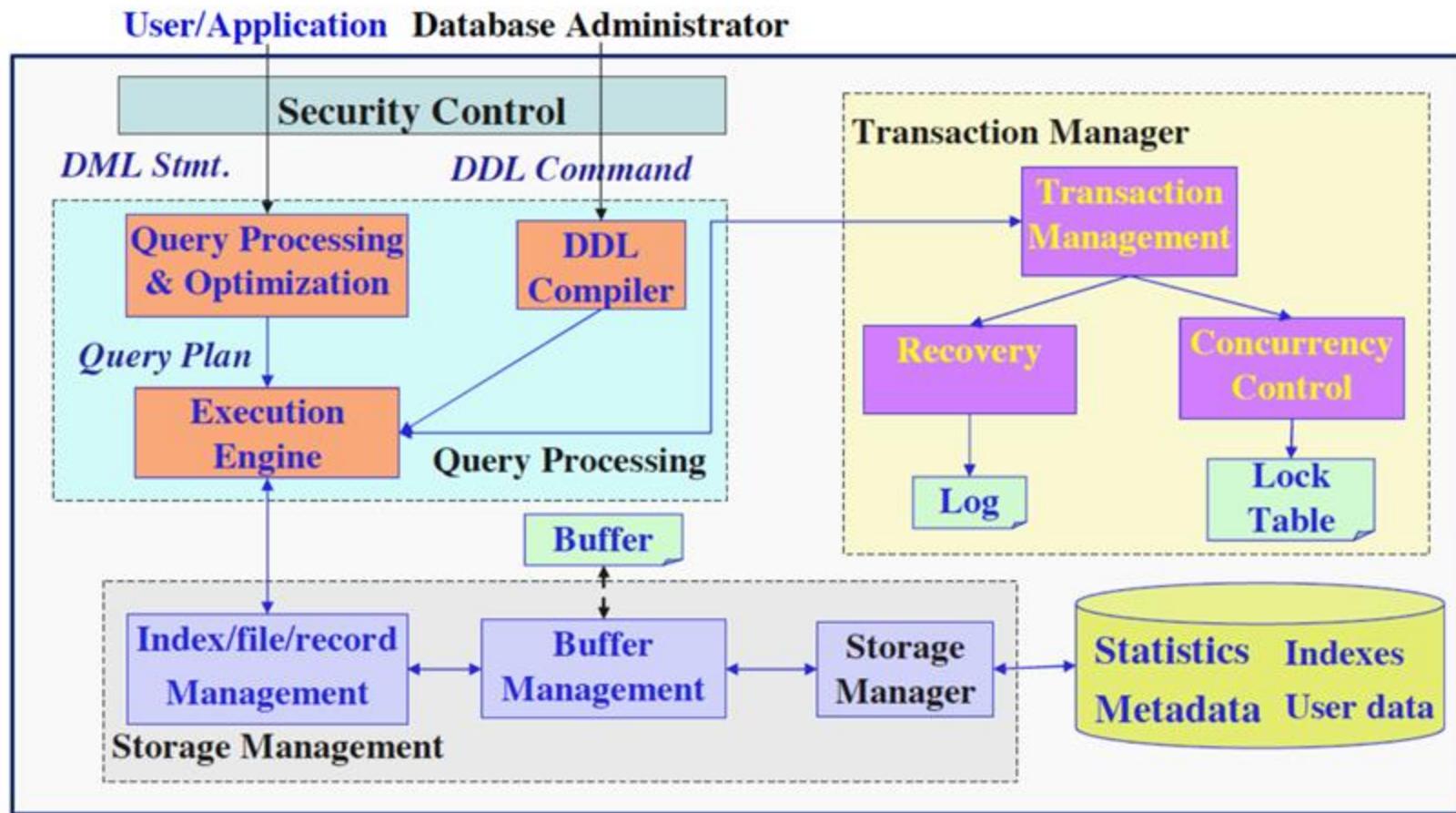


Storing Data: Disk and File

COMP9311 25T3; Week 7

By *Wenjie Zhang, UNSW*

Functional Components of DBMS



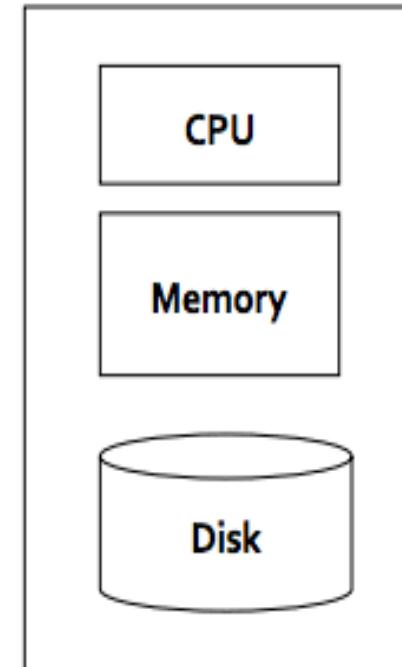
Memory Hierarchy

- ***Primary Storage:*** main memory.
fast access, expensive.
- ***Secondary storage:*** hard disk.
slower access, less expensive.
- ***Tertiary storage:*** tapes, cd, etc.
slowest access, cheapest.

Primary Storage

Main memory:

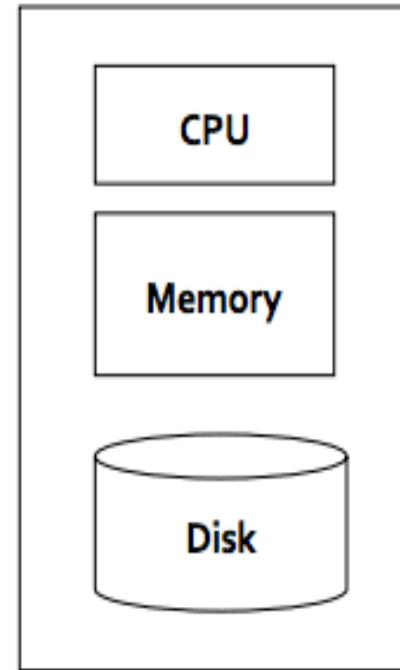
- Fast access (10s to 100s of nanoseconds; 1 nanosecond = 10^{-9} seconds)
- Generally too small (or too expensive) to store the entire database
- **Volatile** — contents of main memory are usually lost if a power failure or system crash occurs.



Secondary Storage

Magnetic-disk

- Data is stored on spinning disk, and read/written magnetically
- Primary medium for the long-term storage of data; typically stores entire database.
- **Data must be moved from disk to main memory for access, and written back for storage**
 - **Much slower access than main memory**
 - **Direct-access** – possible to read data on disk in any order.
 - Survives power failures and system crashes
 - Recall: disk failure can destroy data, but is rare



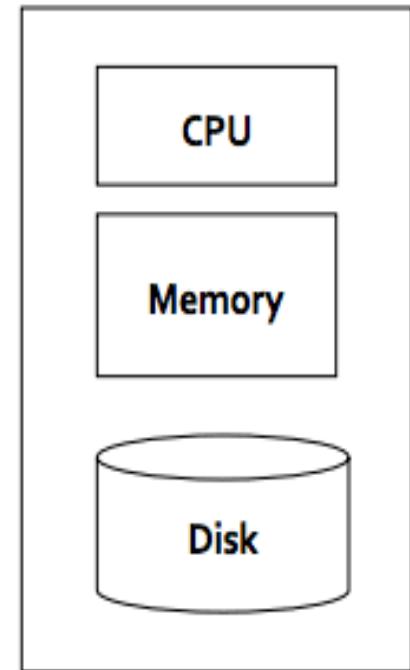
Latency Numbers Every Programmer Should Know

Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access (DRAM, from CPU)	120 ns	6 min
Solid-state disk I/O (flash memory)	50-150 µs	2-6 days
Rotational disk I/O	1-10 ms	1-12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1-3 s	105-317 years

CPU cost vs I/O cost

The implementation issues

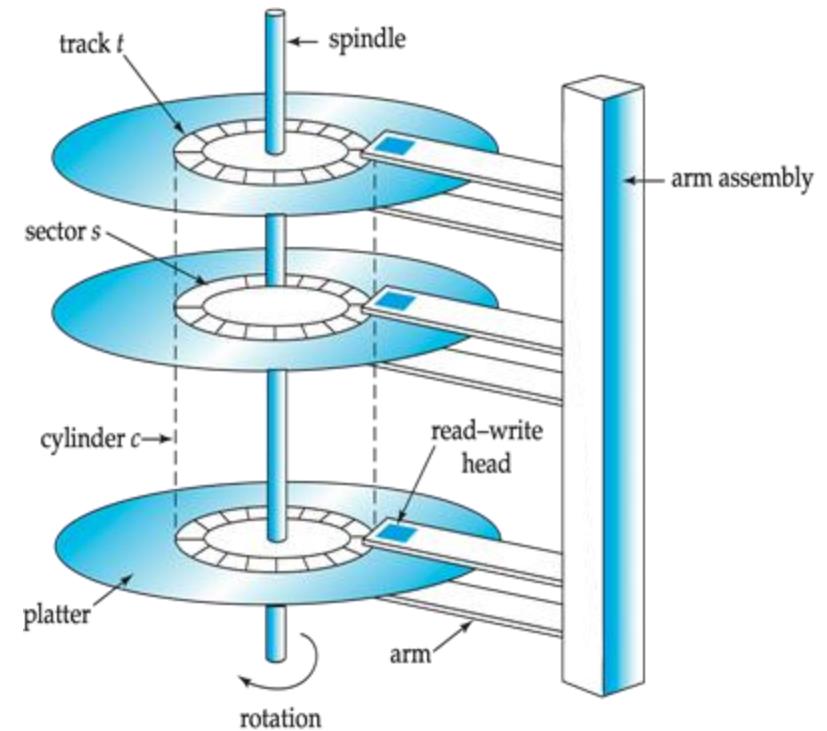
- There are two main costs, CPU cost and I/O (Input/Output) cost.
 - CPU cost is to process data in main memory.
 - I/O cost is to read/write data from/into disk.
- The dominating cost is I/O cost. For query processing in DBMS, CPU cost can be ignored.
- The key issue is to reduce I/O cost.
 - It is to reduce the number of I/O accesses.
- What is I/O cost?
 - A block (or page) to be read/written from/into disk is one I/O access (or one disk-block/page access).



OLD Magnetic Hard Disk

Characteristics of disks:

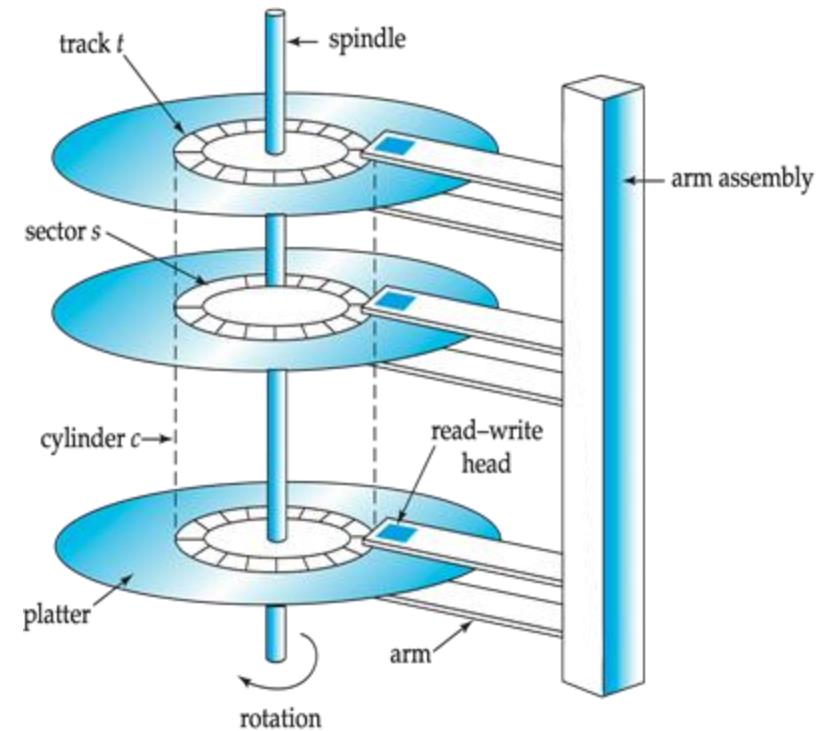
- collection of platters
- each platter = set of tracks
- each track = sequence of sectors (blocks)



NOTE: Diagram simplifies the structure of actual disk drives

OLD Magnetic Hard Disk

- Data must be in memory for the DBMS to operate on it.
- Smallest process unit is **Block**: If a single record in a block is needed, the entire block is transferred.



NOTE: Diagram simplifies the structure of actual disk drives

Disks

Access time includes:

- seek time (find the right track, e.g., 10msec)
- rotational delay (find the right sector, e.g., 5msec)
- transfer time (read/write block, e.g., 10 μ sec)

Random access is dominated by **seek time** and **rotational delay**

Disk Space Management

Improving Disk Access:

Use knowledge of data access patterns.

- E.g., two records often accessed together: put them in the same block (clustering)
- E.g., records scanned sequentially: place them in consecutive sectors on same track

Keep Track of Free Blocks

- Maintain a list of free blocks
- Use bitmap

Using OS File System to Manage Disk Space

- extend OS facilities, but not rely on the OS file system.
- (portability and scalability)

Storage Access

Data must be in memory for the DBMS to operate on it.

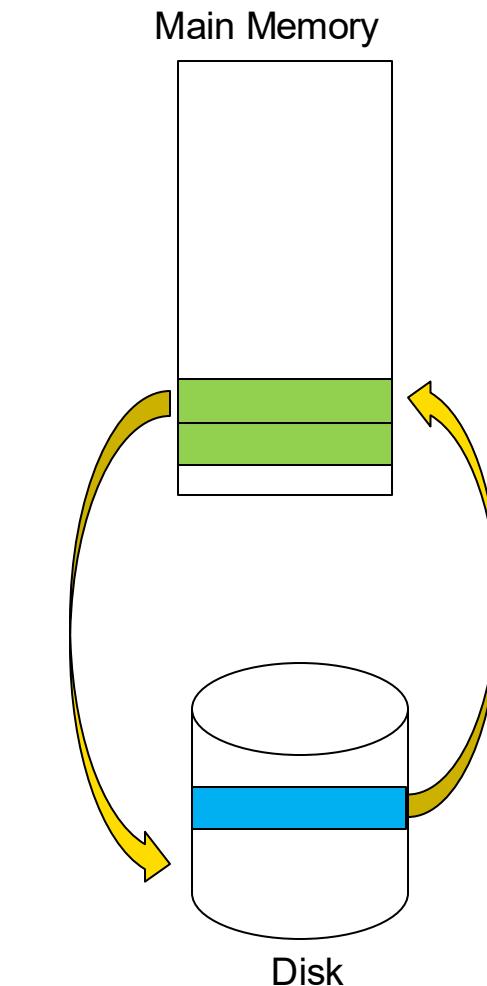
A database file is partitioned into fixed-length storage units called **blocks**. Blocks are units of both storage allocation and data transfer.

Database system seeks to **minimize the number of block transfers** between the disk and memory.

We can reduce the number of disk accesses by keeping as many blocks as possible in main memory.

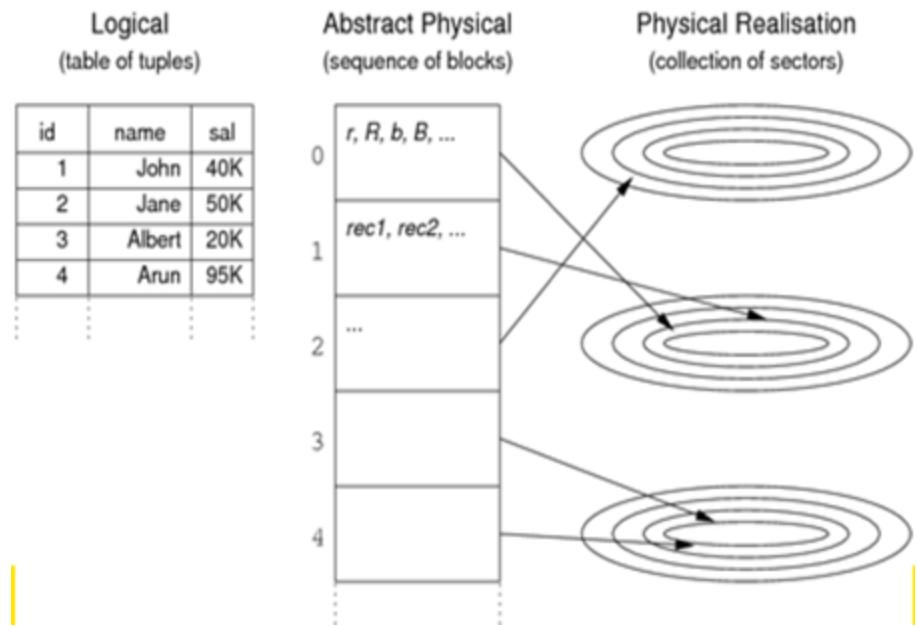
Buffer – portion of main memory available to store copies of disk blocks.

Buffer manager – subsystem responsible for allocating buffer space in main memory.

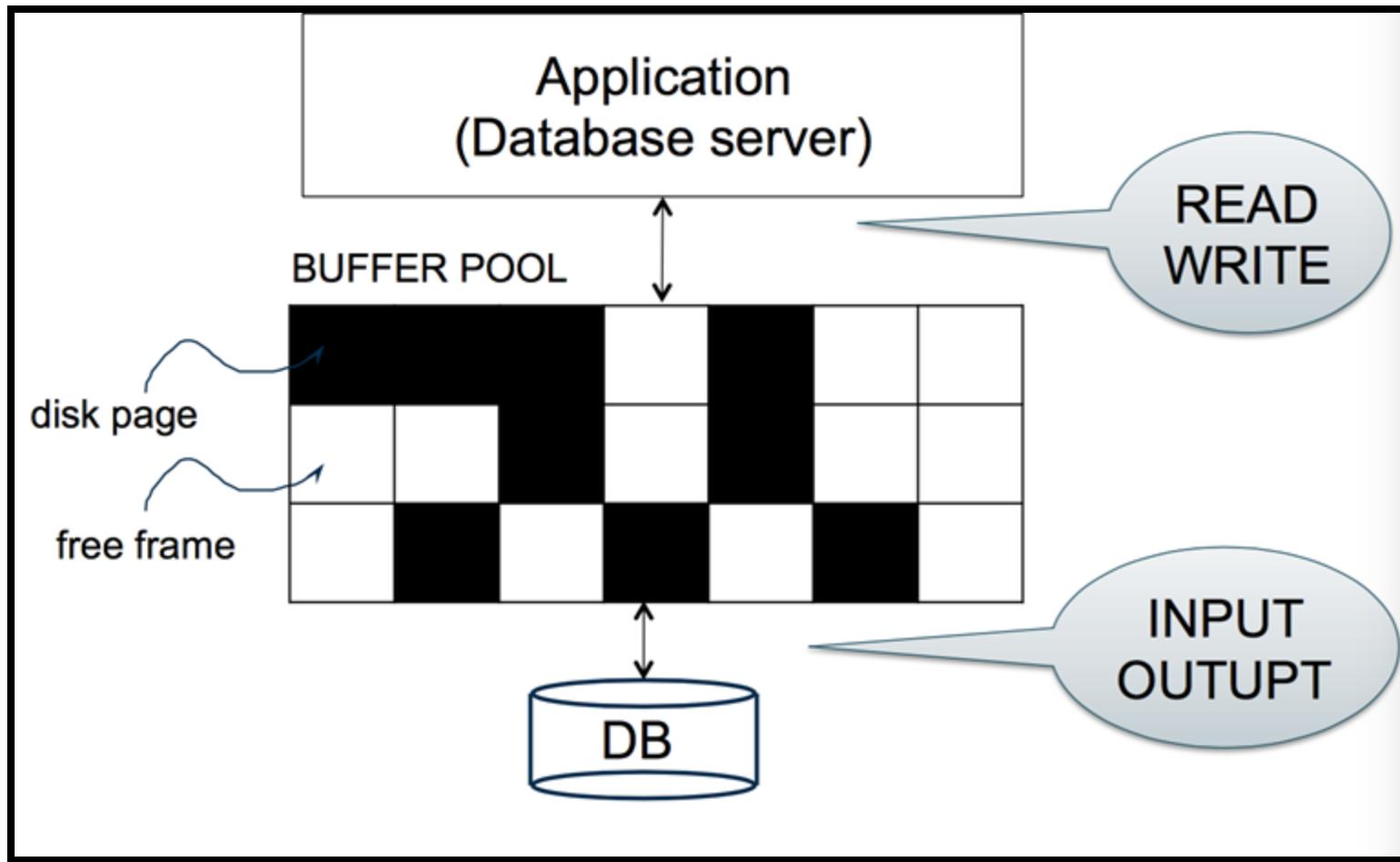


Disk-Block Access

- Smallest process unit is a **block**: If a single record in a block is needed, the entire block is transferred.
- Data are transferred between disk and main memory in **units** of blocks.
- A relation is stored as a **file on disk**.
- A file is a sequence of blocks, where a **block** is a fixed-length storage unit.
- A block is also called a **page**.



Buffer Management in a DBMS



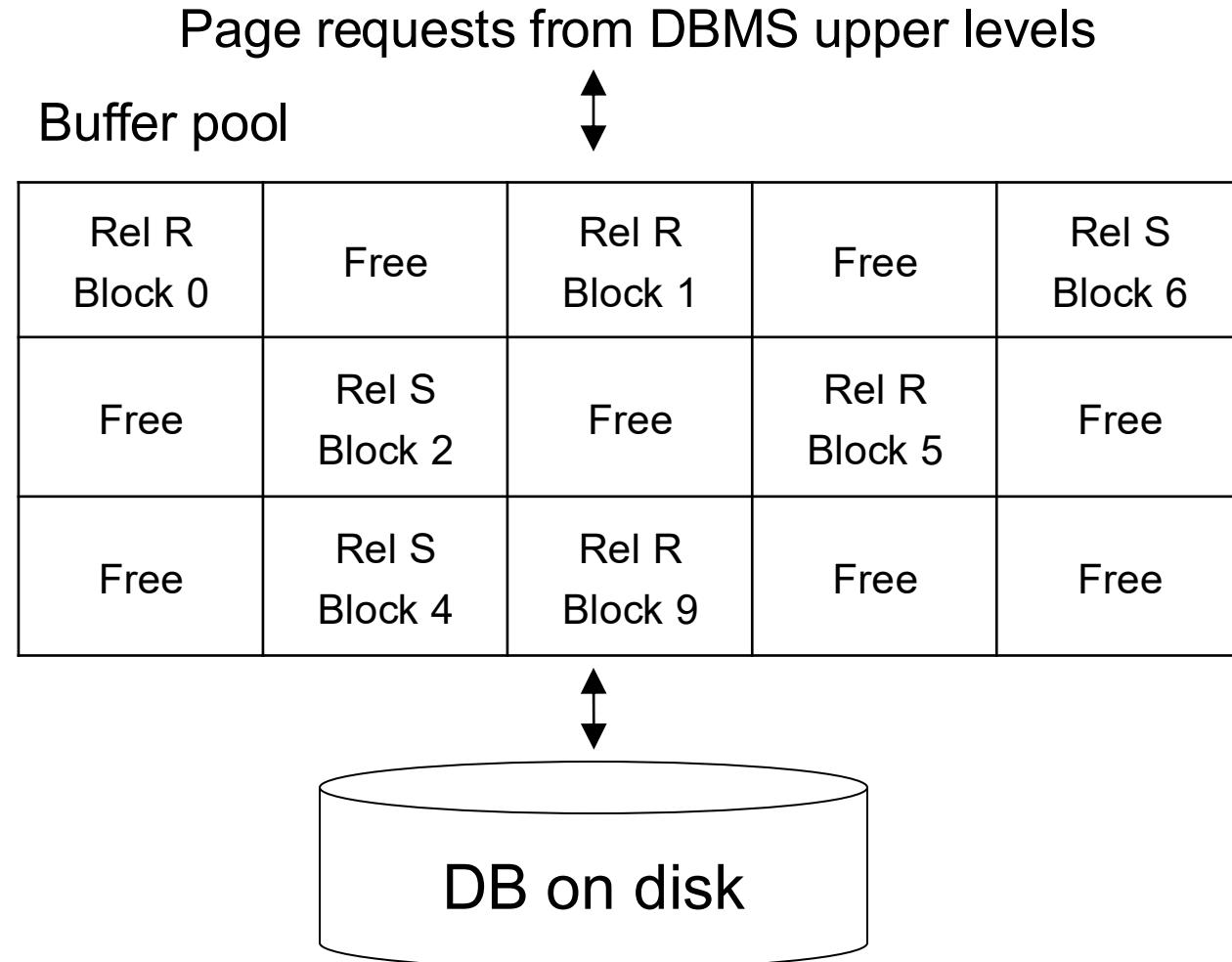
Buffer Management

Manages traffic between disk and memory by maintaining a **buffer pool** in main memory.

Buffer Pool

- collection of page slots (frames) which can be filled with copies of disk block data.
- E.g., One page = 4096 Bytes = One block

Buffer Pool



Buffer Pool

The **request_block** operation

If block **is** already in buffer pool:

- no need to read it again
- use the copy there (unless write-locked)

If block **is not** in buffer pool yet:

- need to read from hard disk into a free frame
- if no free frames, need to remove block using a **buffer replacement policy**.

The **release_block** function indicates that block is no longer in use

- good candidate for removal / replacing

Buffer Pool

For each frame, we need to know:

- whether it is currently in use
- whether it has been modified since loading (*dirty bit*)
- how many transactions are currently using it (*pin count*)
- (maybe) time-stamp for most recent access

Buffer Pool

The *release_block* operation

- Decrement pin count for specified page.
- No real effect until replacement required.

The *write_block* operation

- Updates contents of page in pool
- Set dirty bit on
- Note: Doesn't actually write to disk, until been replaced, or forced to commit

The *force_block* operation

- "commits" by writing to disk.

Buffer Replacement Policies

Least Recently Used (LRU)

- release the frame that has not been used for the longest period.
- intuitively appealing idea but can perform badly

Most Recently Used (MRU):

- release the frame used most recently

First in First Out (FIFO)

- need to maintain a queue of frames
- enter tail of queue when read in

Random

No one is guaranteed to be better than the others.

Quite dependent on applications.

Quiz 1:

Example1:

Data pages: P1, P2, P3, P4

Queries:

Q1: read P1; Q2: read P2;

Q3: read P3; Q4: read P1;

Q5: read P2;

Buffer:

P1	Q1		
----	----	--	--

P1	Q1	P2	Q2	
----	----	----	----	--

P1	Q1	P2	Q2	P3	Q3
----	----	----	----	----	----

P1	Q4	P2	Q2	P3	Q3
----	----	----	----	----	----

P1	Q4	P2	Q5	P3	Q3
----	----	----	----	----	----



Quiz 1:

Example1:

Data pages: P1, P2, P3, P4

Queries:

Q1: read P1; Q2: read P2;

Q3: read P3; Q4: read P1;

Q5: read P2;

Buffer:

P1 Q4	P2 Q5	P3 Q3
-------	-------	-------

How about if Q6 read P4?
Using different buffer replacement policies

Quiz 1(LRU):

Example1:

Data pages: P1, P2, P3, P4

Queries:

Q1: read P1; Q2: read P2;

Q3: read P3; Q4: read P1;

Q5: read P2;

Buffer:

P1 Q4	P2 Q5	P3 Q3
-------	-------	-------



How about if Q6 read P4?
Using different buffer replacement policies
LRU: Least Recently Used

P1 Q4	P2 Q5	P4 Q6
-------	-------	--------------



Quiz 1(MRU):

Example1:

Data pages: P1, P2, P3, P4

Queries:

Q1: read P1; Q2: read P2;

Q3: read P3; Q4: read P1;

Q5: read P2;

Buffer:

P1 Q4	P2 Q5	P3 Q3
-------	-------	-------



P1 Q4	P4 Q6	P3 Q3
-------	--------------	-------



How about if Q6 read P4?
Using different buffer replacement policies
MRU: Most Recently Used

Quiz 1(FIFO):

Example1:

Data pages: P1, P2, P3, P4

Queries:

Q1: read P1; Q2: read P2;

Q3: read P3; Q4: read P1;

Q5: read P2;

Buffer:

P1 Q4	P2 Q5	P3 Q3
-------	-------	-------



P4 Q6	P2 Q5	P3 Q3
-------	-------	-------



How about if Q6 read P4?
Using different buffer replacement policies
FIFO: First In First Out

Quiz 1(Random):

Example1:

Data pages: P1, P2, P3, P4

Queries:

Q1: read P1; Q2: read P2;

Q3: read P3; Q4: read P1;

Q5: read P2;

Buffer:

P1 Q4	P2 Q5	P3 Q3
-------	-------	-------



How about if Q6 read P4?
Using different buffer replacement policies
Random

Randomly choose one buffer to replace

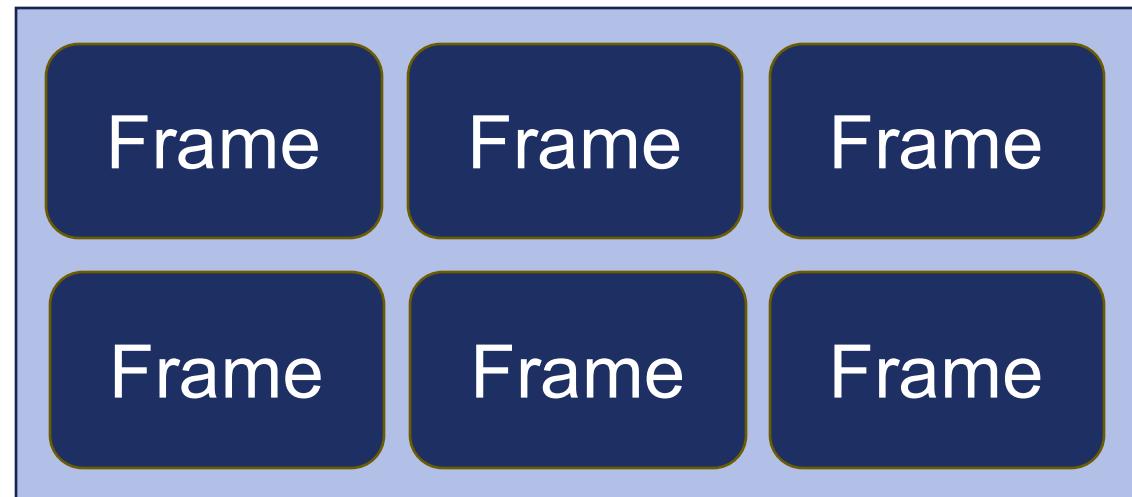
Cache Performance

- Cache hits
 - pages can be served by the cache
- Cache misses
 - pages have to be retrieved from the disk
- **Hit rate = #cache hits / (#cache hits + #cache misses)**

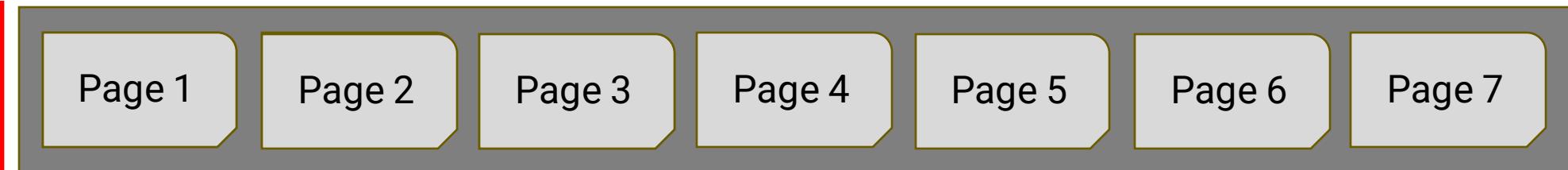
Repeated Scan (LRU)

Cache Hit: 0

Attempts: 0



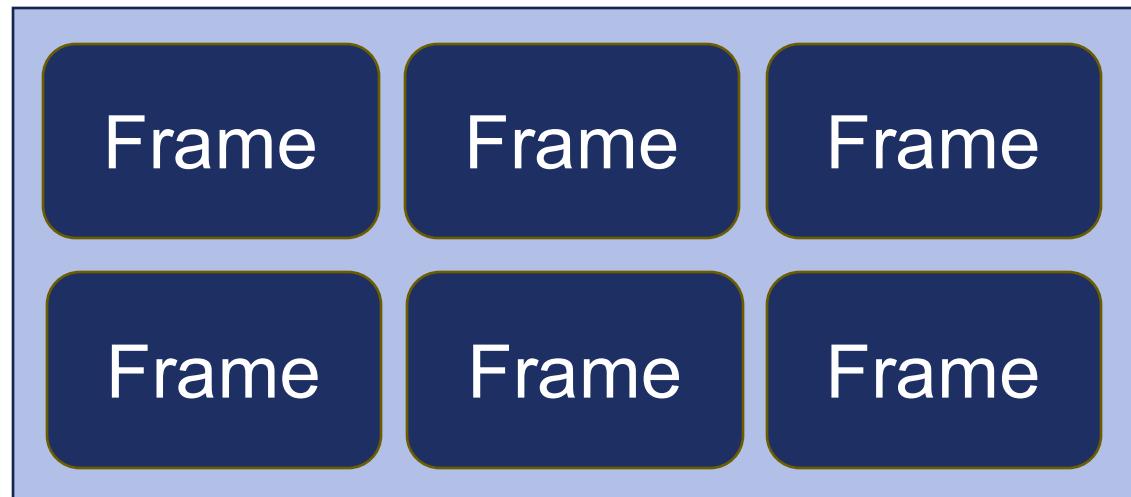
Disk Space Manager



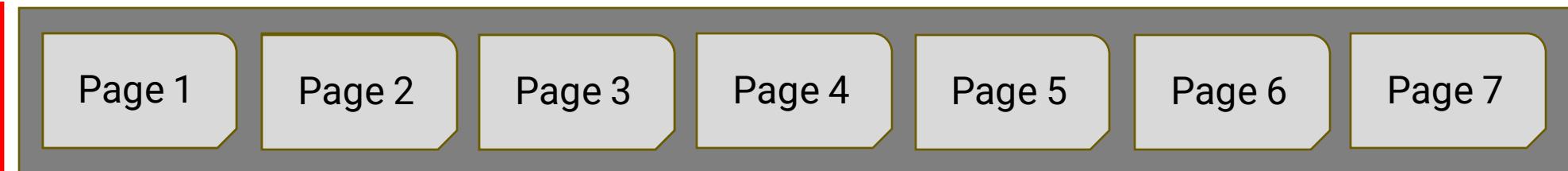
Repeated Scan (LRU): Read Page 1

Cache Hit: 0

Attempts: 1



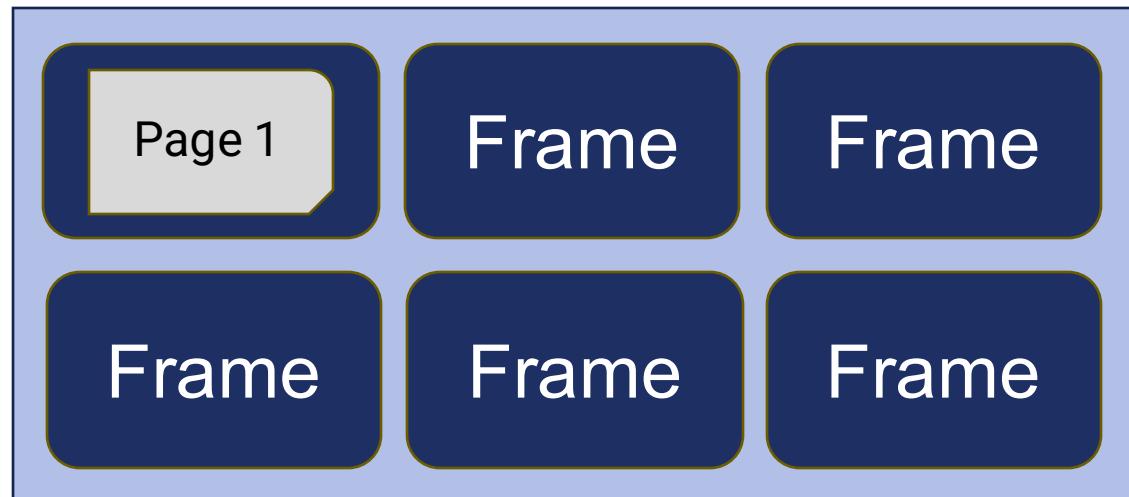
Disk Space Manager



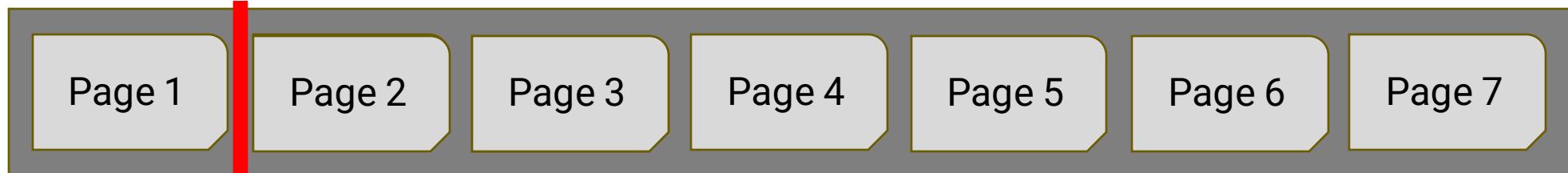
Repeated Scan (LRU): Read Page 2

Cache Hit: 0

Attempts: 2



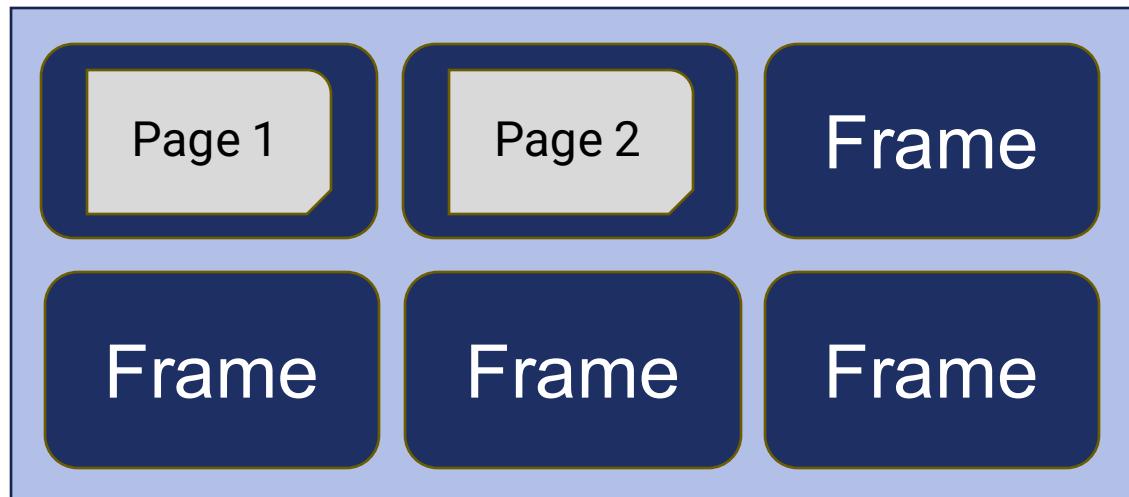
Disk Space Manager



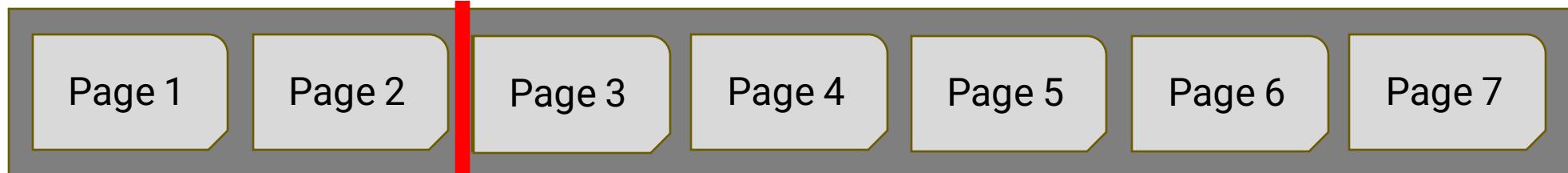
Repeated Scan (LRU): Read Page 3

Cache Hit: 0

Attempts: 3



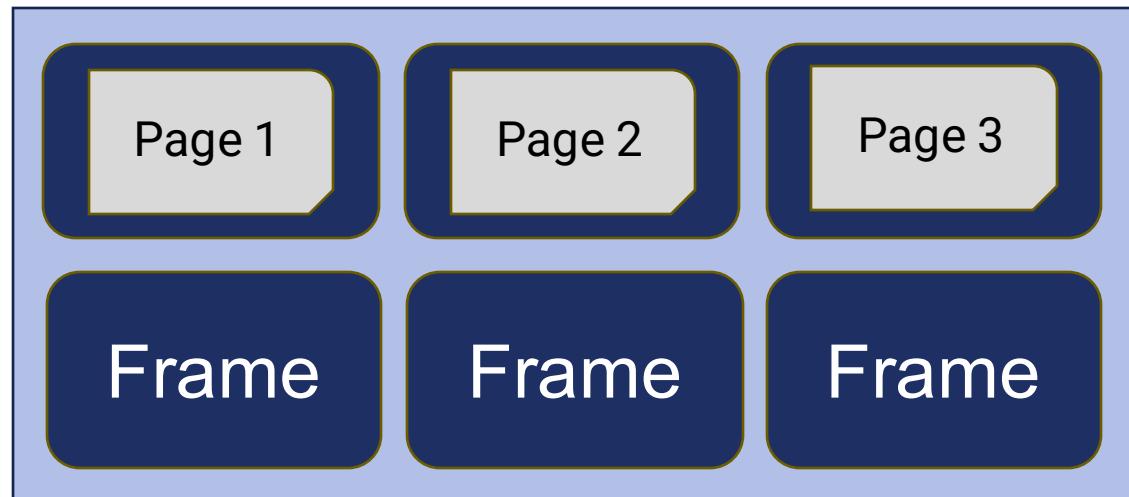
Disk Space Manager



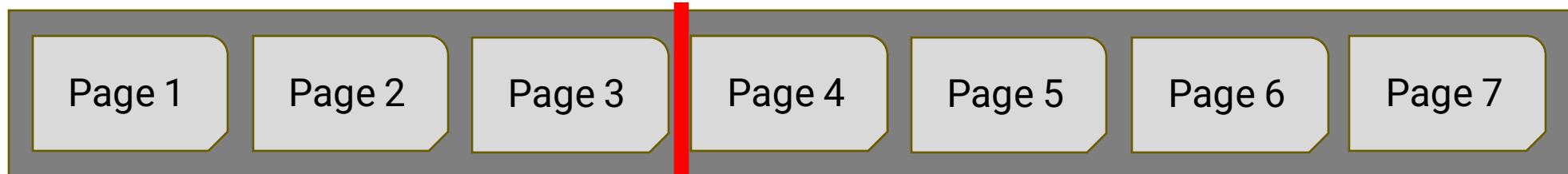
Repeated Scan (LRU): Read Page 4

Cache Hit: 0

Attempts: 4



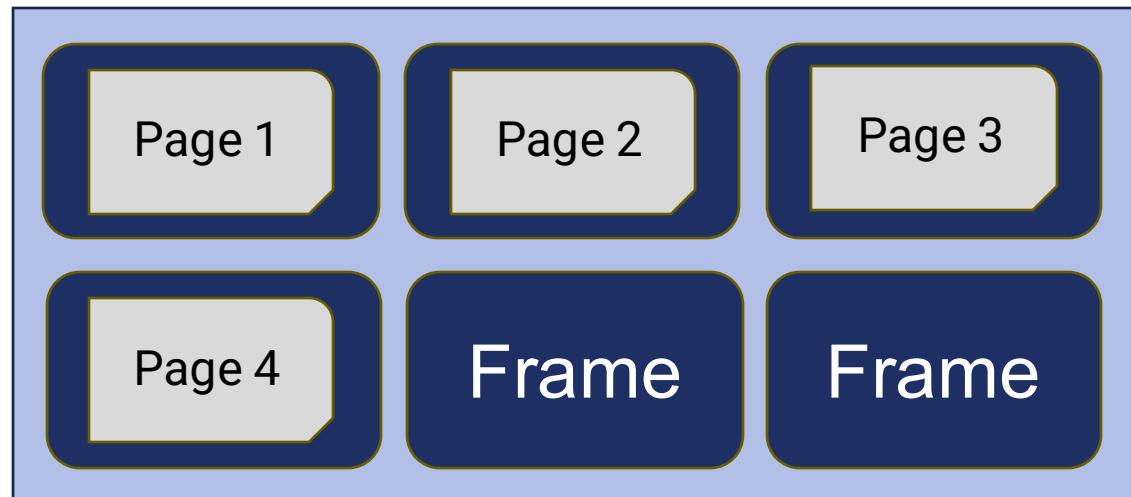
Disk Space Manager



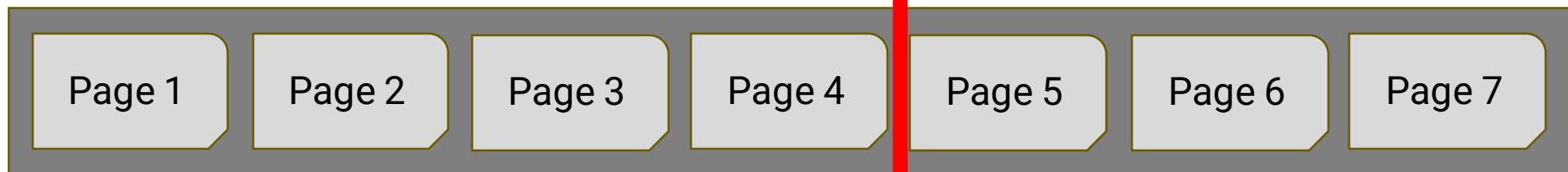
Repeated Scan (LRU): Read Page 5

Cache Hit: 0

Attempts: 5



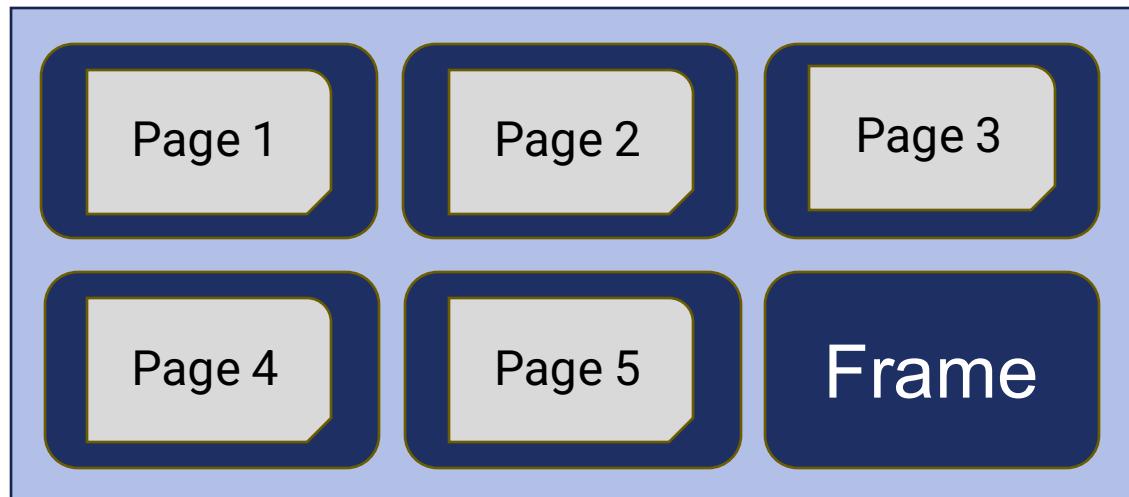
Disk Space Manager



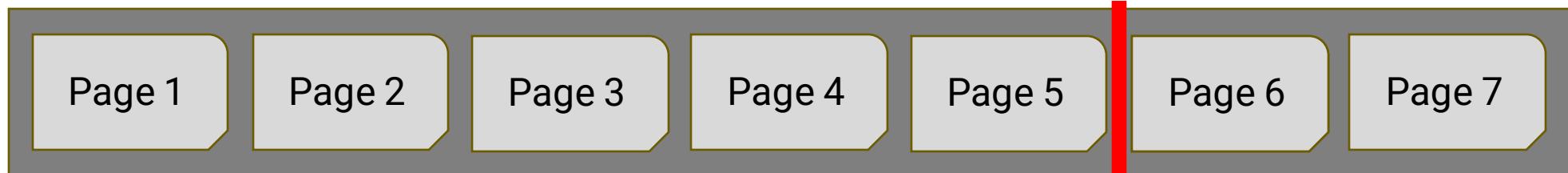
Repeated Scan (LRU): Read Page 6

Cache Hit: 0

Attempts: 6



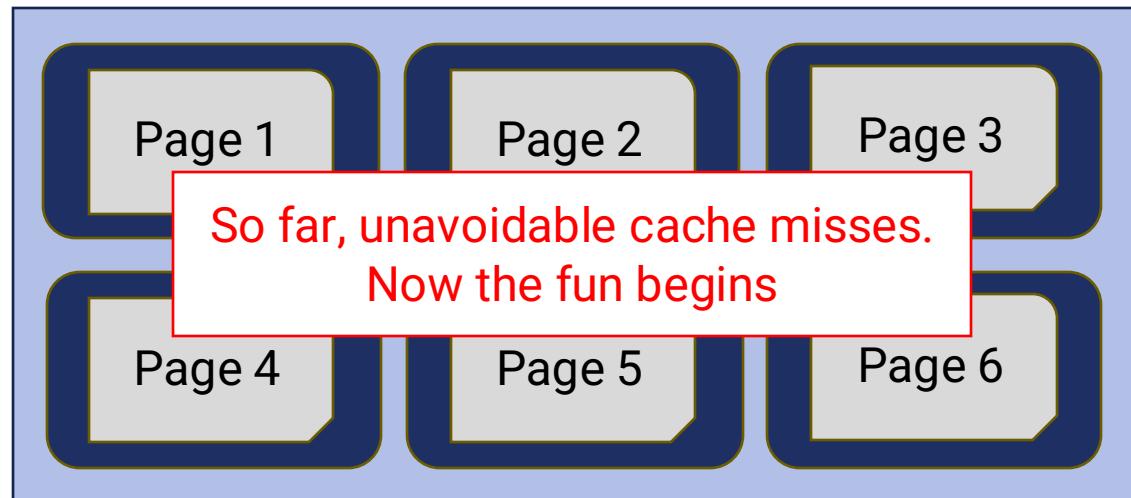
Disk Space Manager



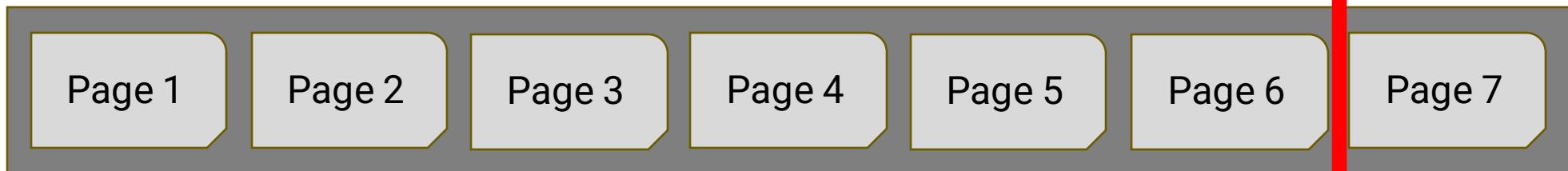
Repeated Scan (LRU): Read Page 6

Cache Hit: 0

Attempts: 6



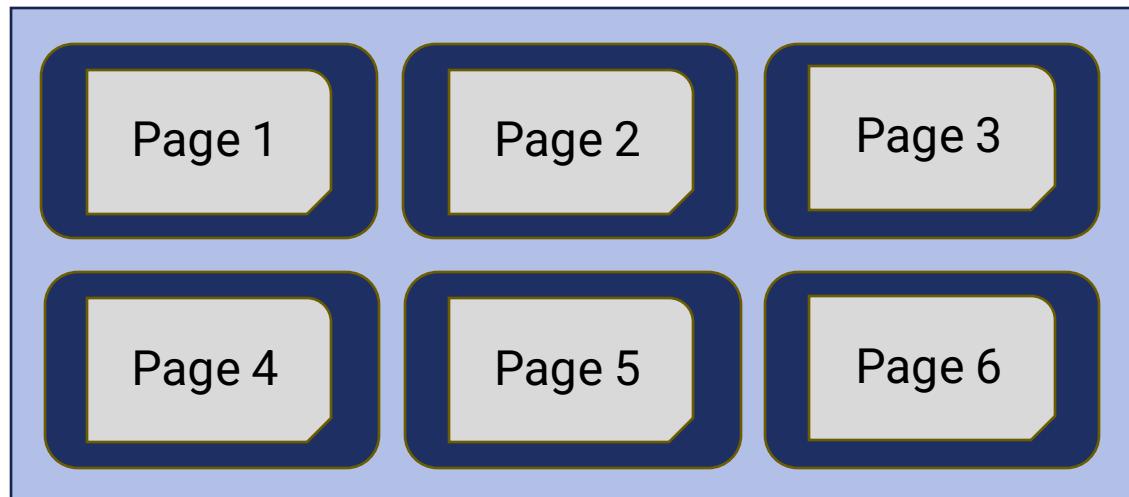
Disk Space Manager



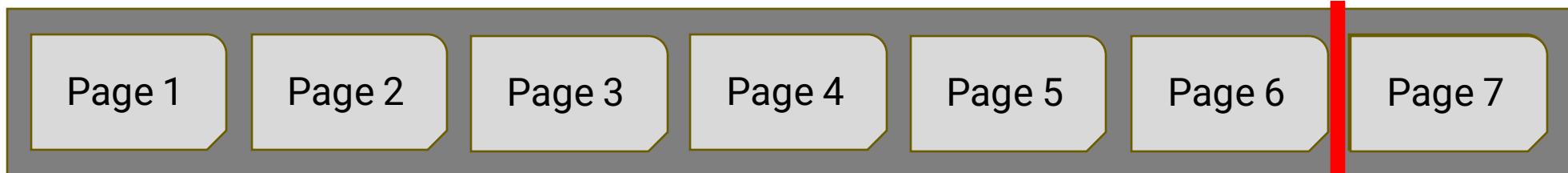
Repeated Scan (LRU): Read Page 7

Cache Hit: 0

Attempts: 7



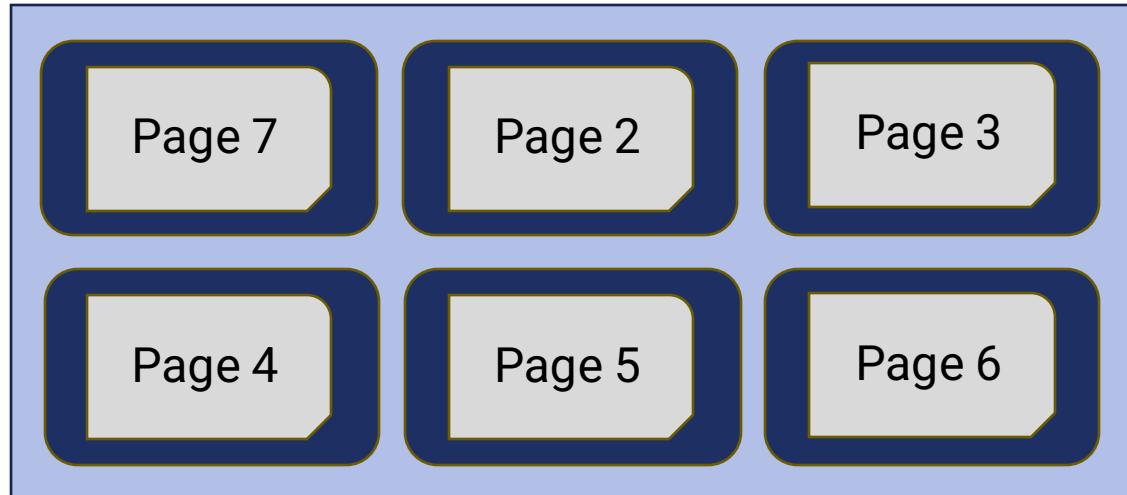
Disk Space Manager



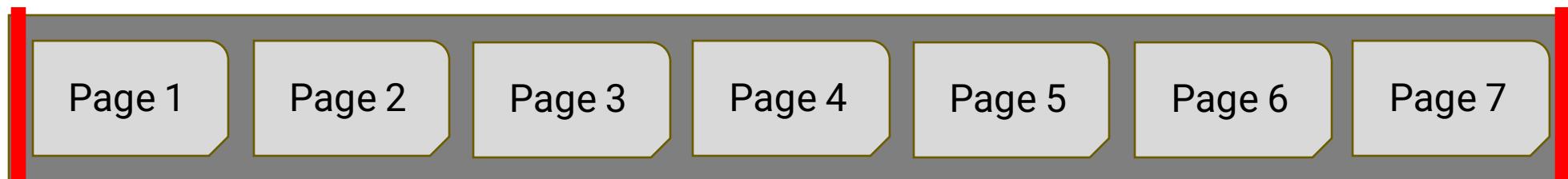
Repeated Scan (LRU): Reset to beginning

Cache Hit: 0

Attempts: 7



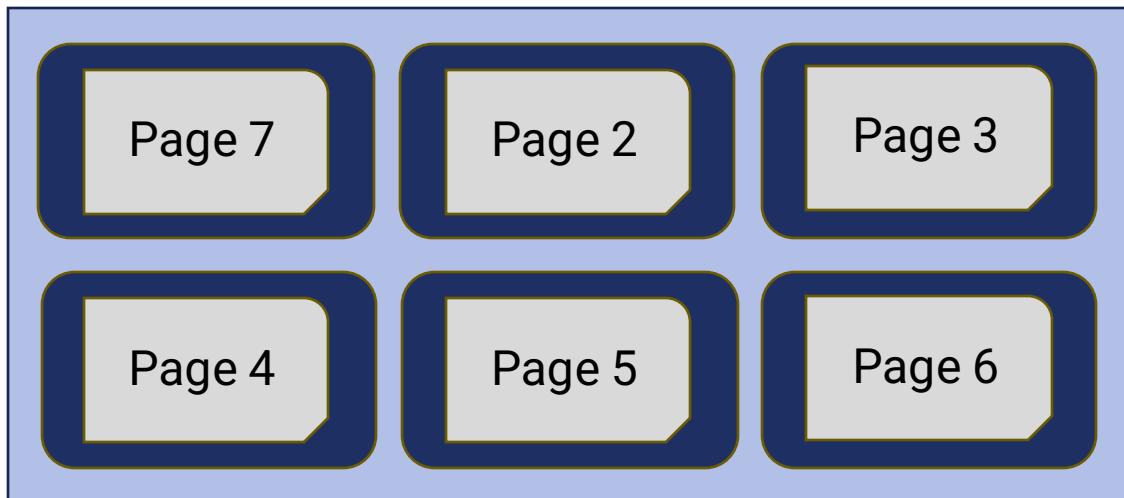
Disk Space Manager



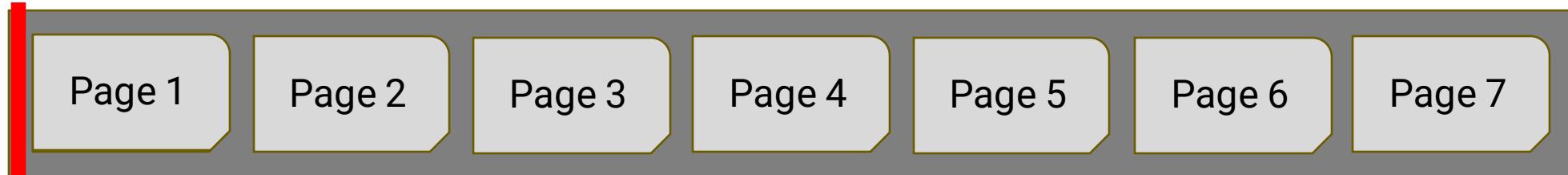
Repeated Scan (LRU): Read Page 1(again)

Cache Hit: 0

Attempts: 8



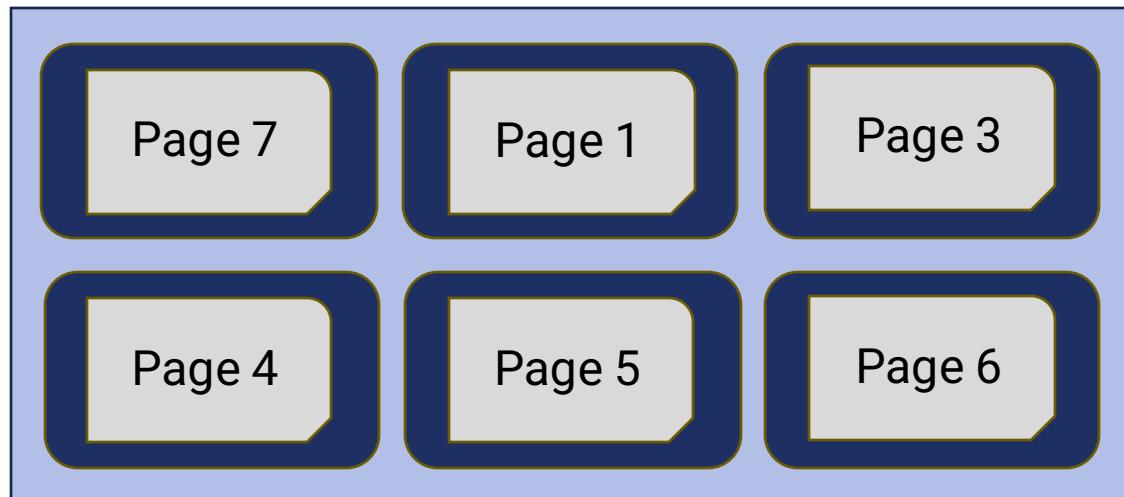
Disk Space Manager



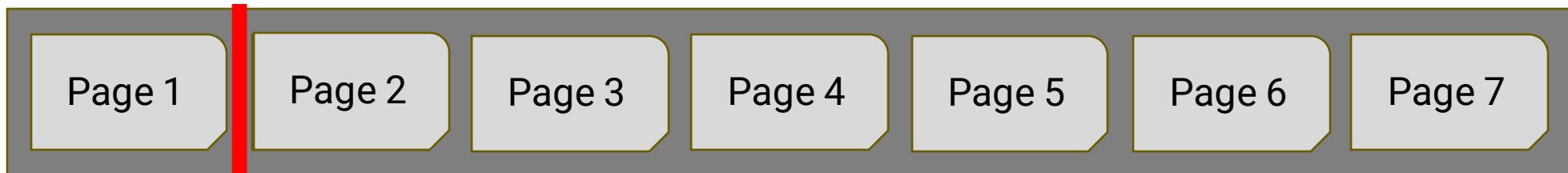
Repeated Scan (LRU): Read Page 2(again)

Cache Hit: 0

Attempts: 9



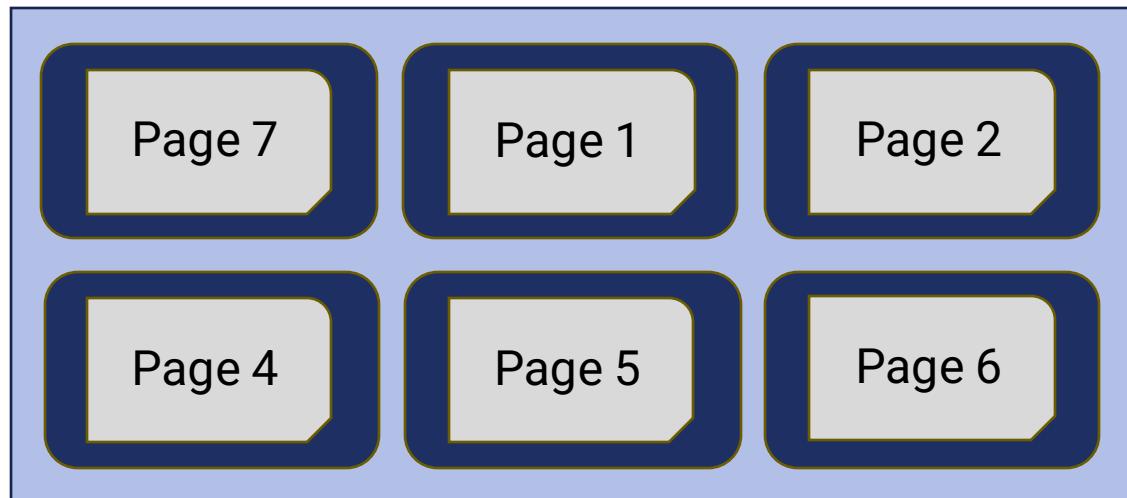
Disk Space Manager



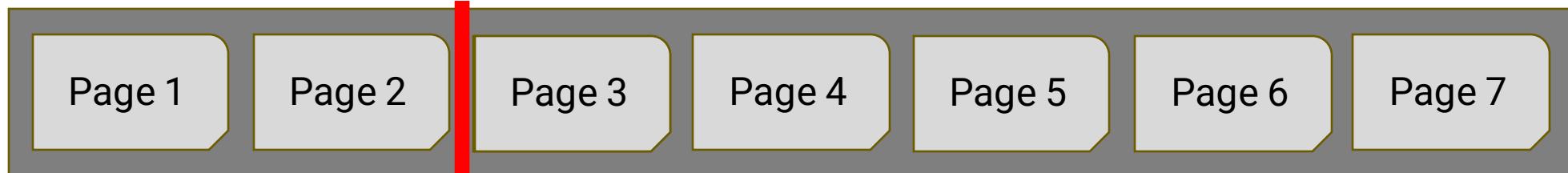
Repeated Scan (LRU): Read Page 3(again)

Cache Hit: 0

Attempts: 10



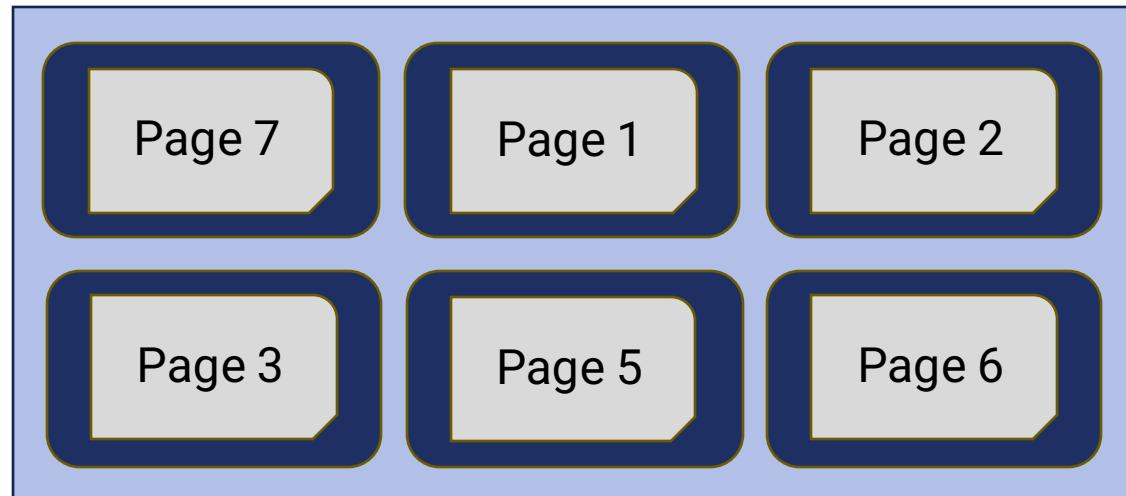
Disk Space Manager



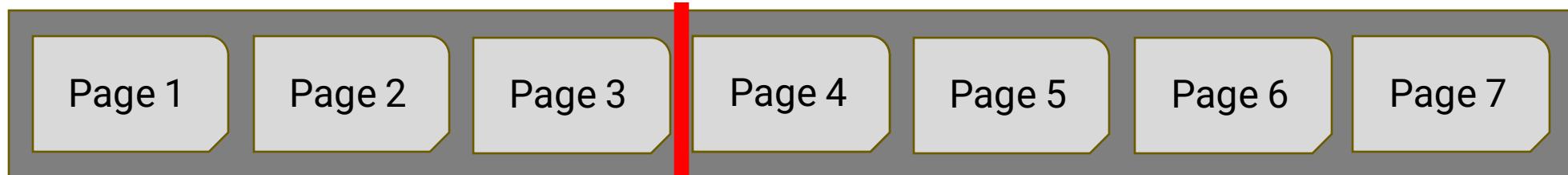
Repeated Scan (LRU): Read Page 4(again)

Cache Hit: 0

Attempts: 11



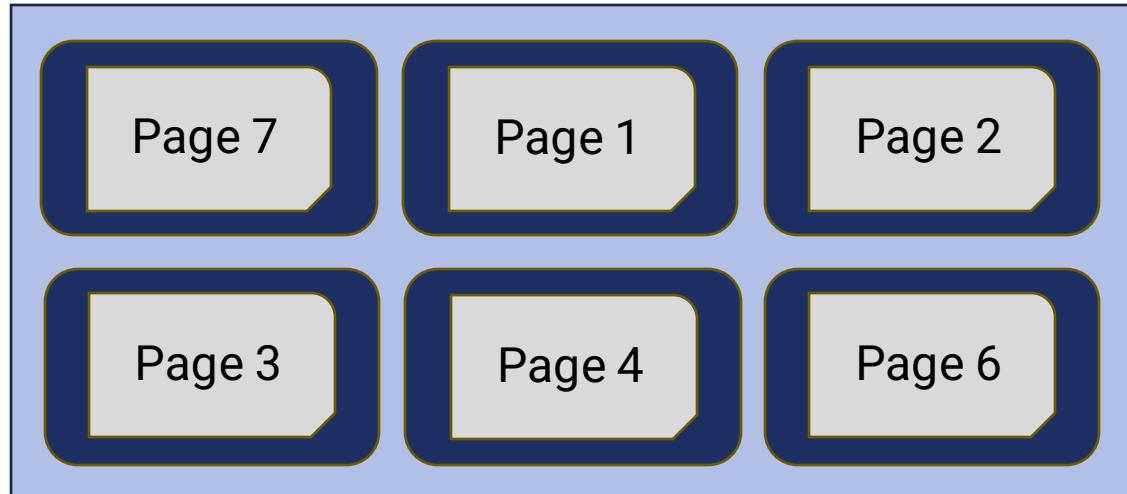
Disk Space Manager



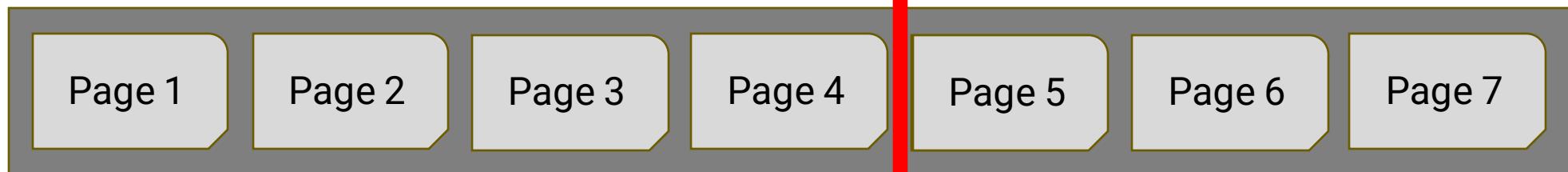
Repeated Scan (LRU): Read Page 5, cont

Cache Hit: 0

Attempts: 12



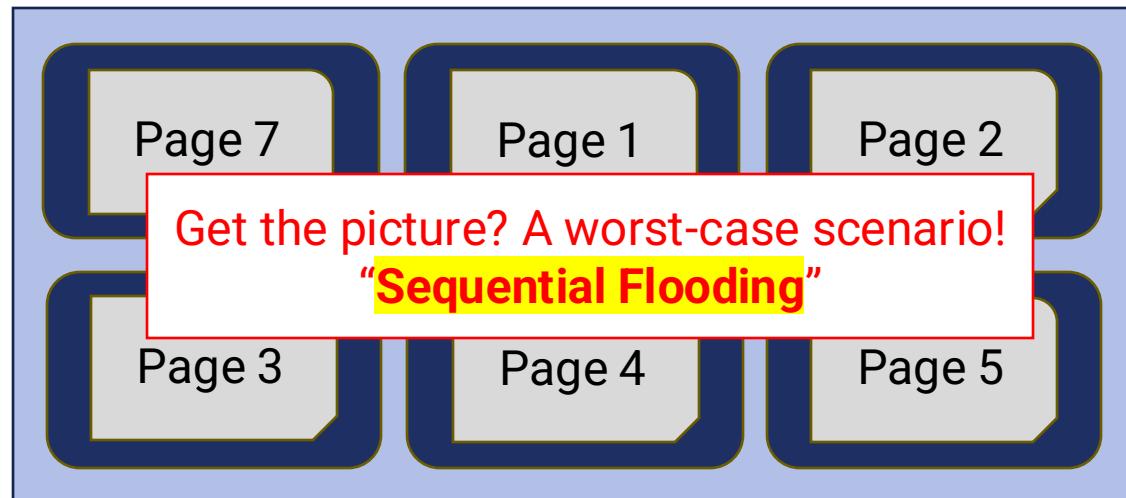
Disk Space Manager



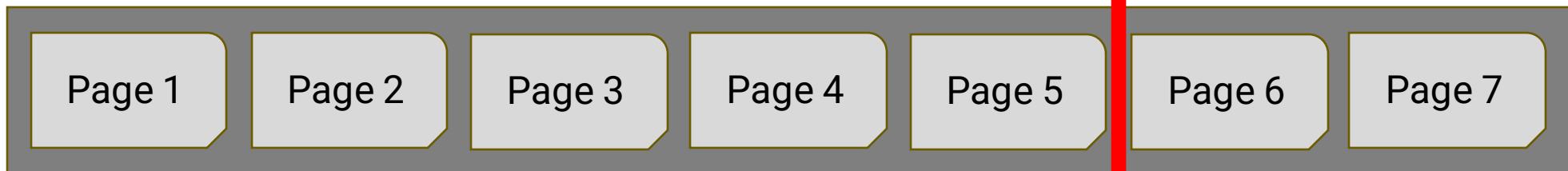
Repeated Scan (LRU): Read Page 5, cont

Cache Hit: 0

Attempts: 12



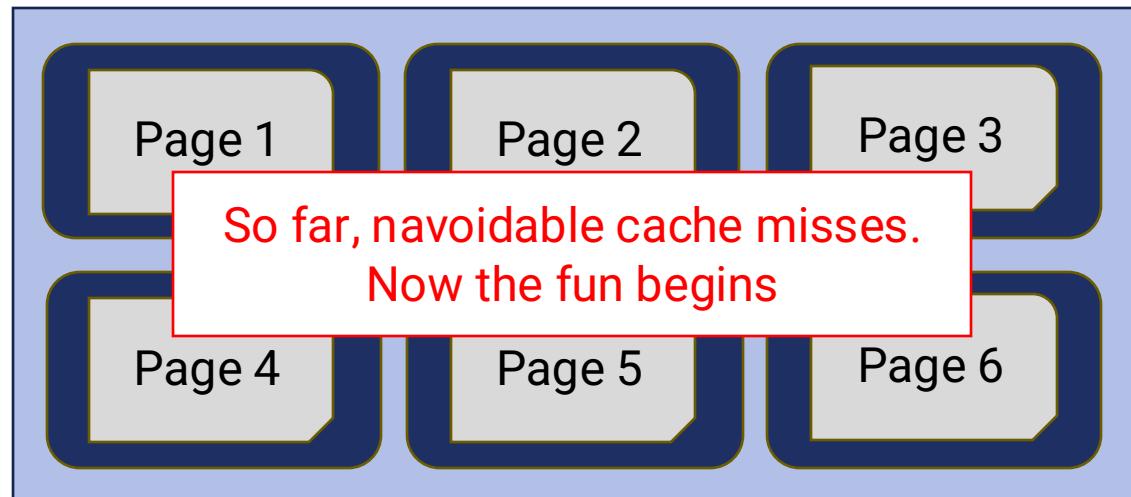
Disk Space Manager



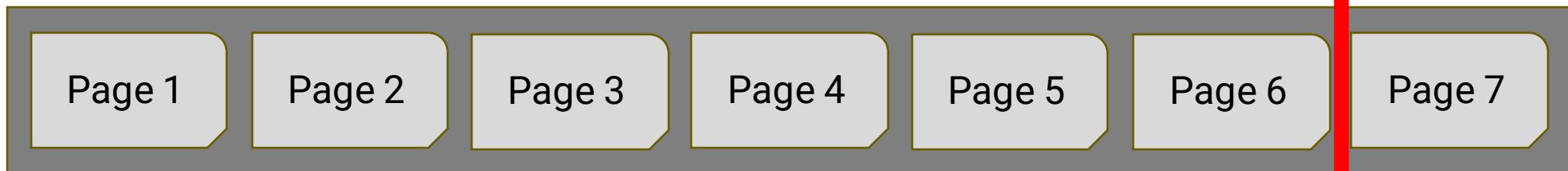
Repeated Scan (MRU)

Cache Hit: 0

Attempts: 6



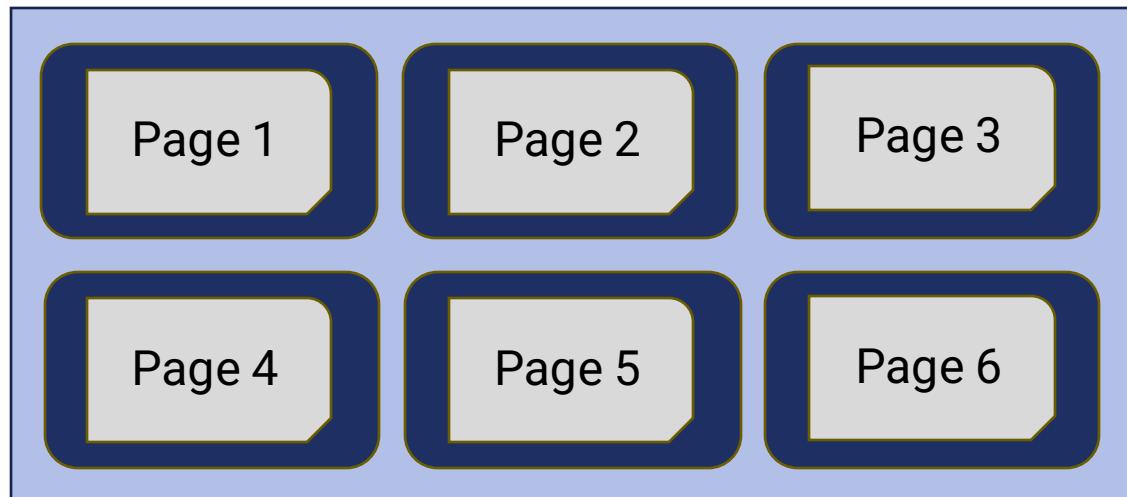
Disk Space Manager



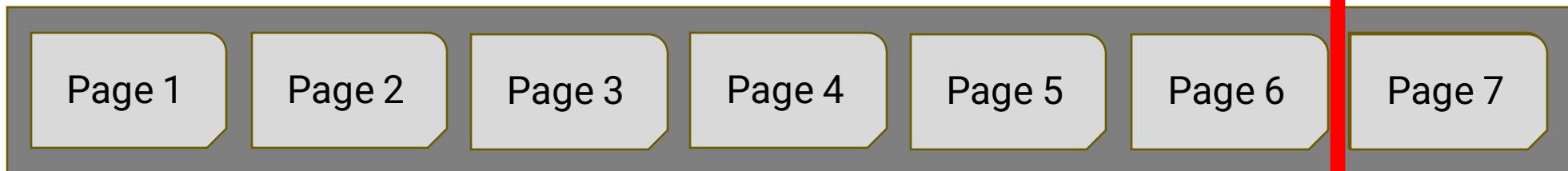
Repeated Scan (MRU): Read Page 7

Cache Hit: 0

Attempts: 7



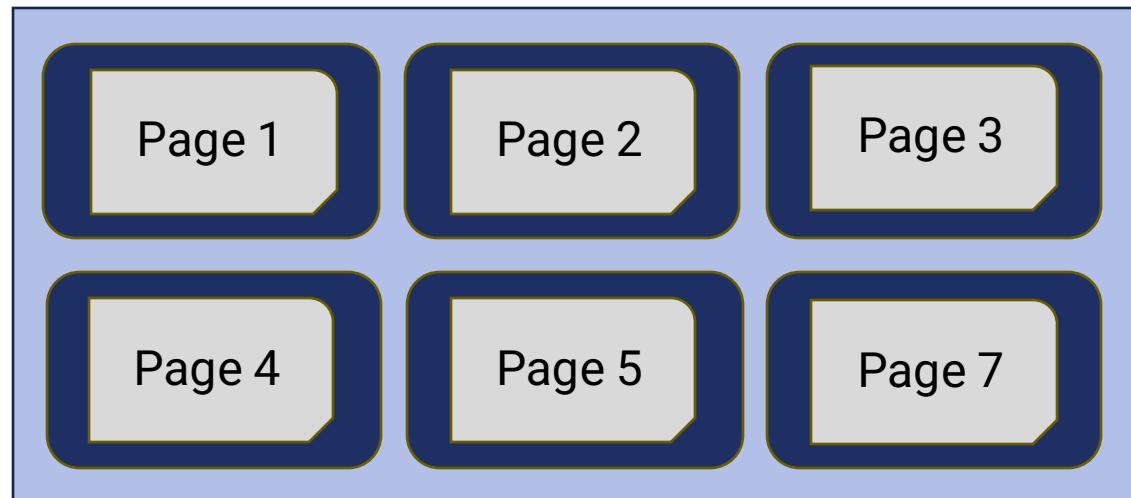
Disk Space Manager



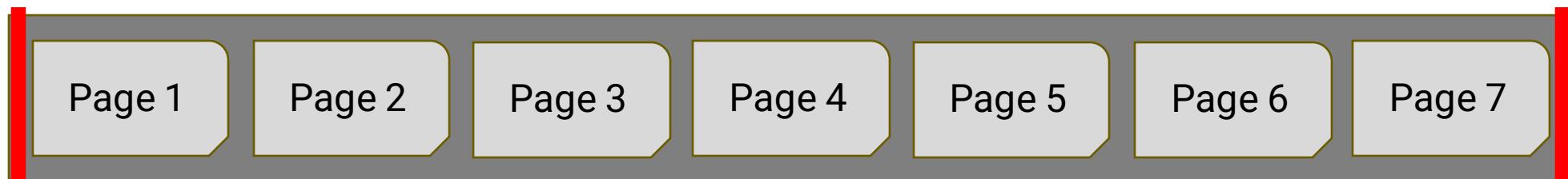
Repeated Scan (MRU): Reset

Cache Hit: 0

Attempts: 7



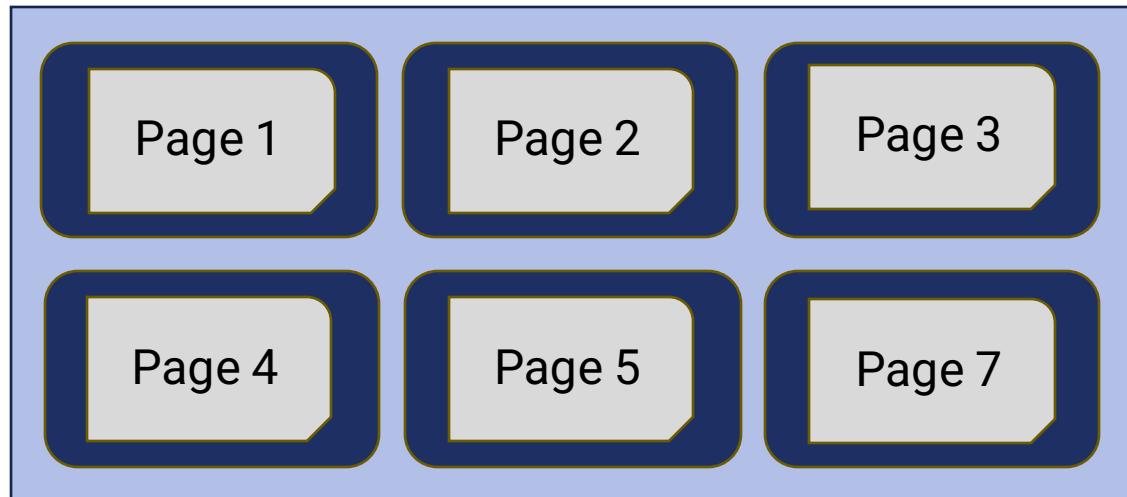
Disk Space Manager



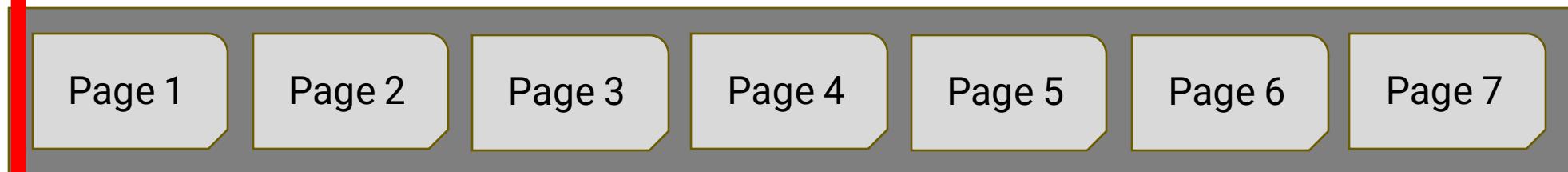
Repeated Scan (MRU): Read Page 1(again)

Cache Hit: 1

Attempts: 8



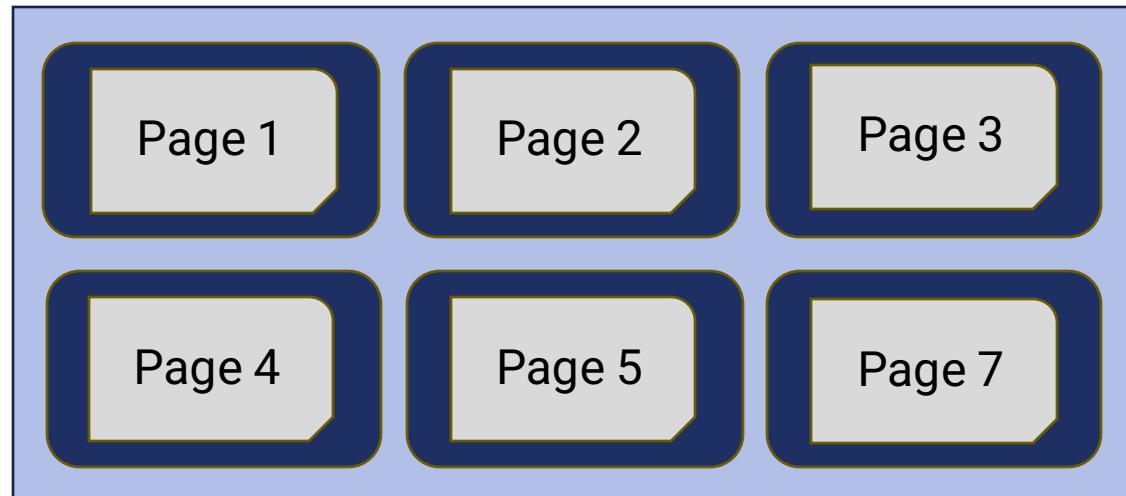
Disk Space Manager



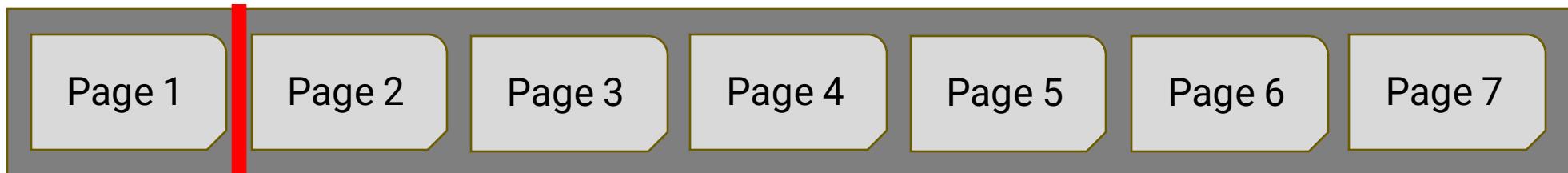
Repeated Scan (MRU): Read Page 2(again)

Cache Hit: 2

Attempts: 9



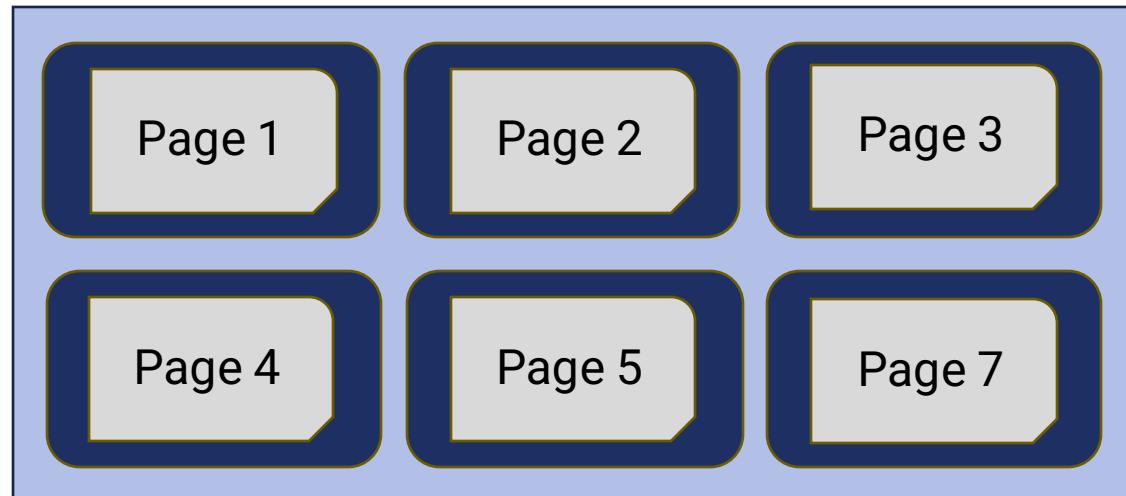
Disk Space Manager



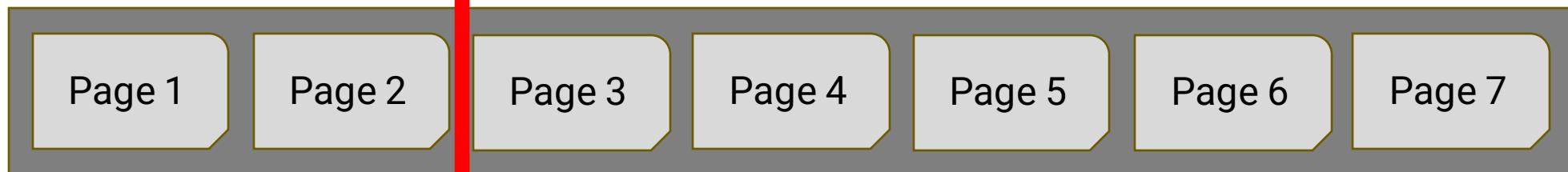
Repeated Scan (MRU): Read Page 3(again)

Cache Hit: 3

Attempts: 10



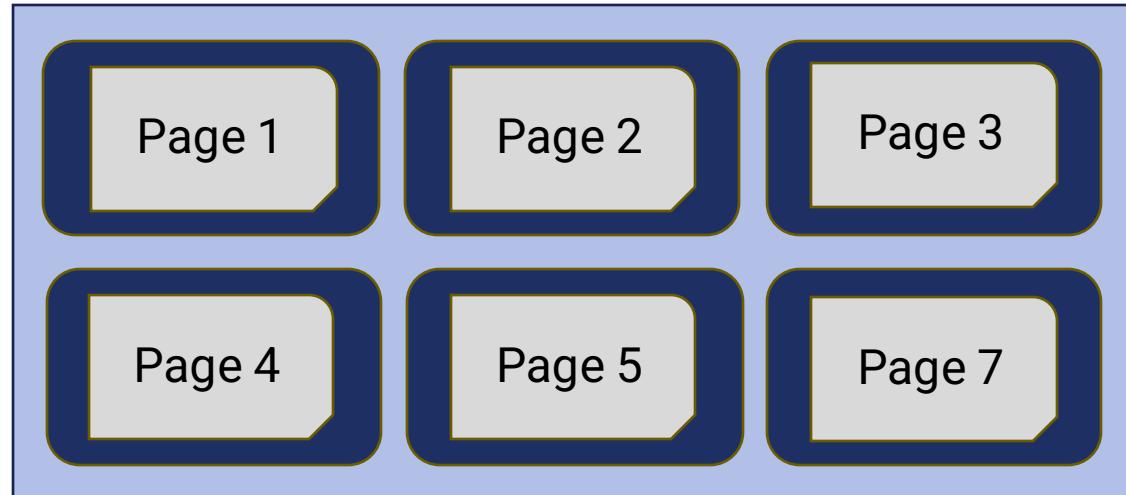
Disk Space Manager



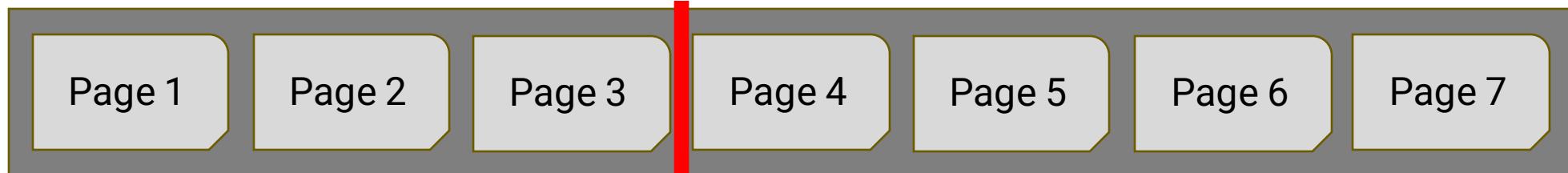
Repeated Scan (MRU): Read Page 4(again)

Cache Hit: 4

Attempts: 11



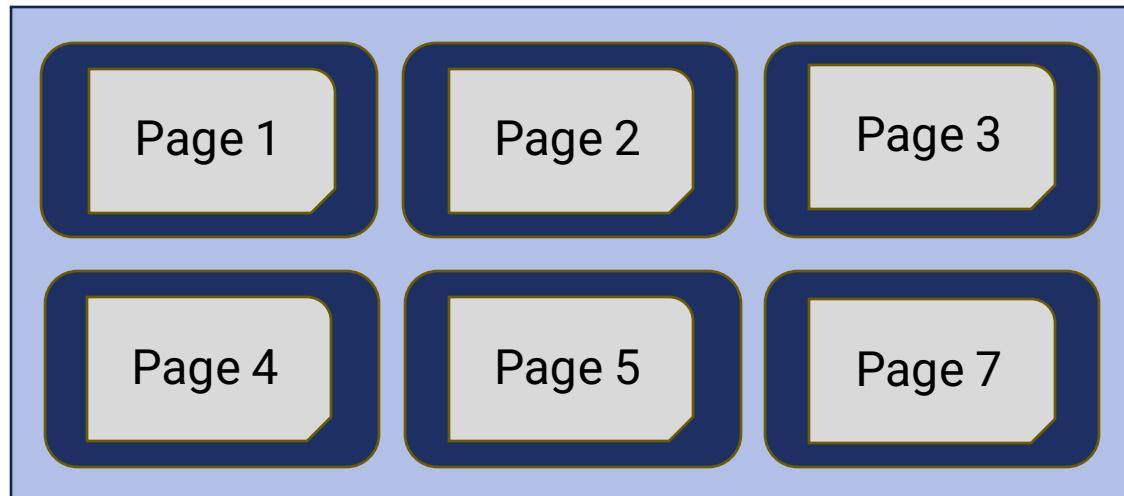
Disk Space Manager



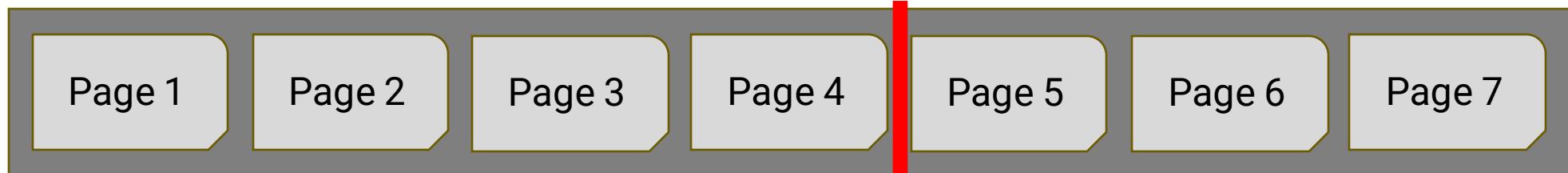
Repeated Scan (MRU): Read Page 5 (again)

Cache Hit: 5

Attempts: 12



Disk Space Manager



Compare LRU and MRU

When LRU and MRU both read Page 5 again

LRU:

Cache hit: 0

Attempts: 12

MRU:

Cache hit: 5

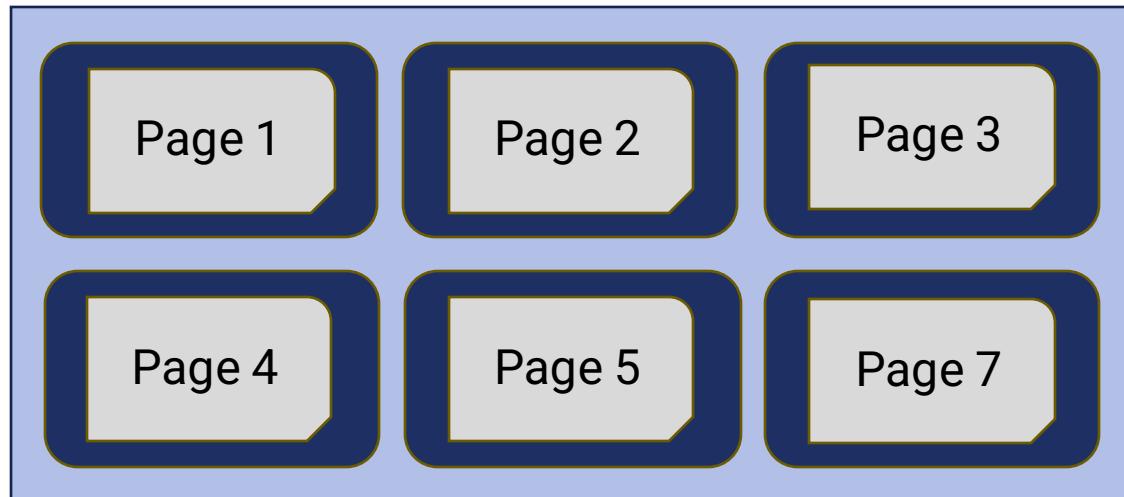
Attempts: 12

What if we keep reading the next page with MRU?

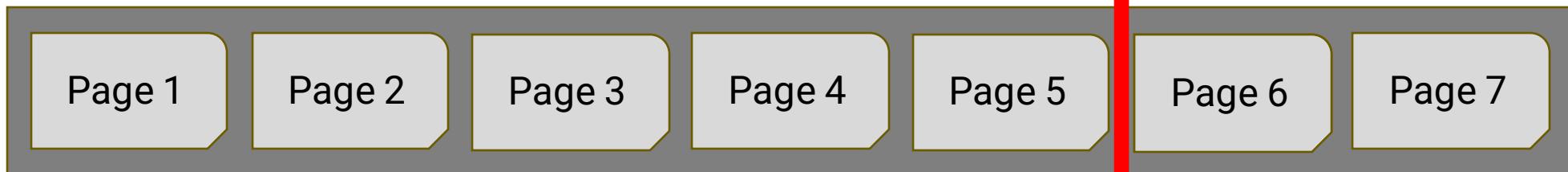
Repeated Scan (MRU): Read Page 6 (again)

Cache Hit: 5

Attempts: 13



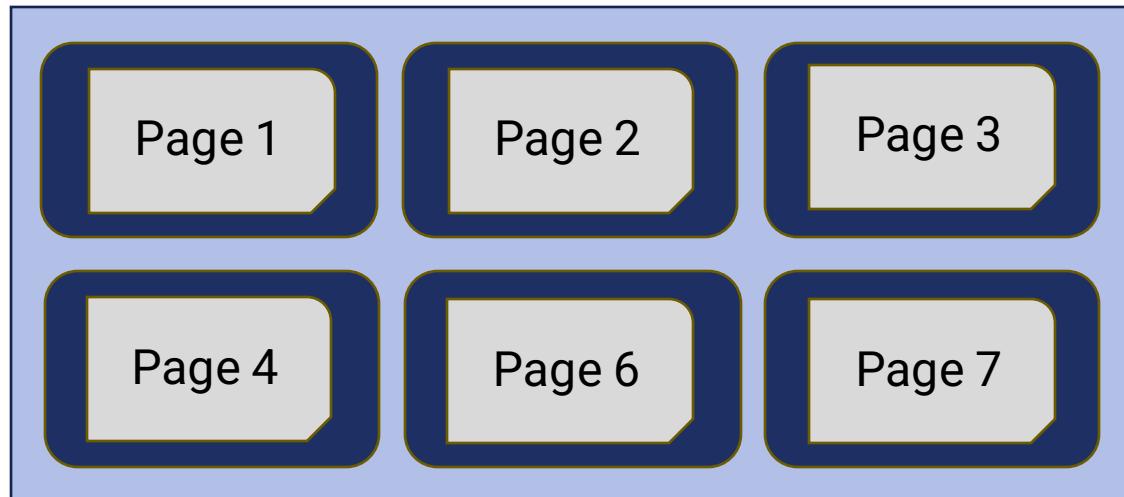
Disk Space Manager



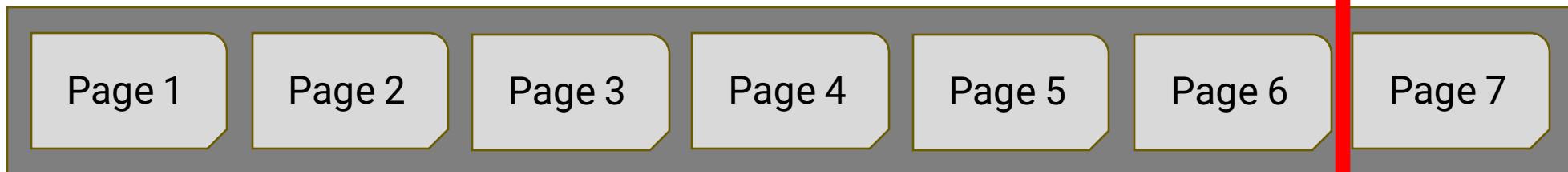
Repeated Scan (MRU): Read Page 7 (again)

Cache Hit: 6

Attempts: 14



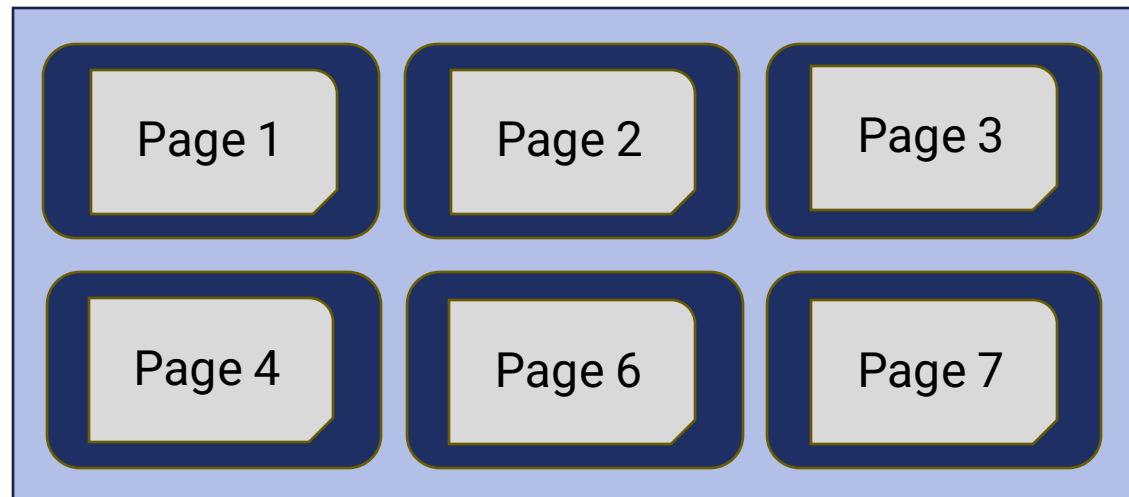
Disk Space Manager



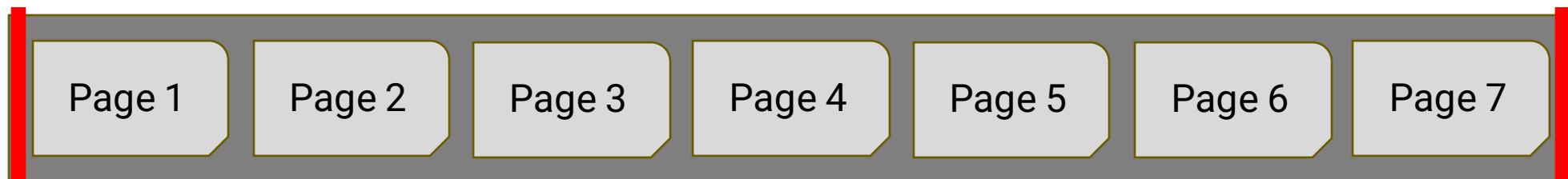
Repeated Scan (MRU): Reset (again)

Cache Hit: 6

Attempts: 14



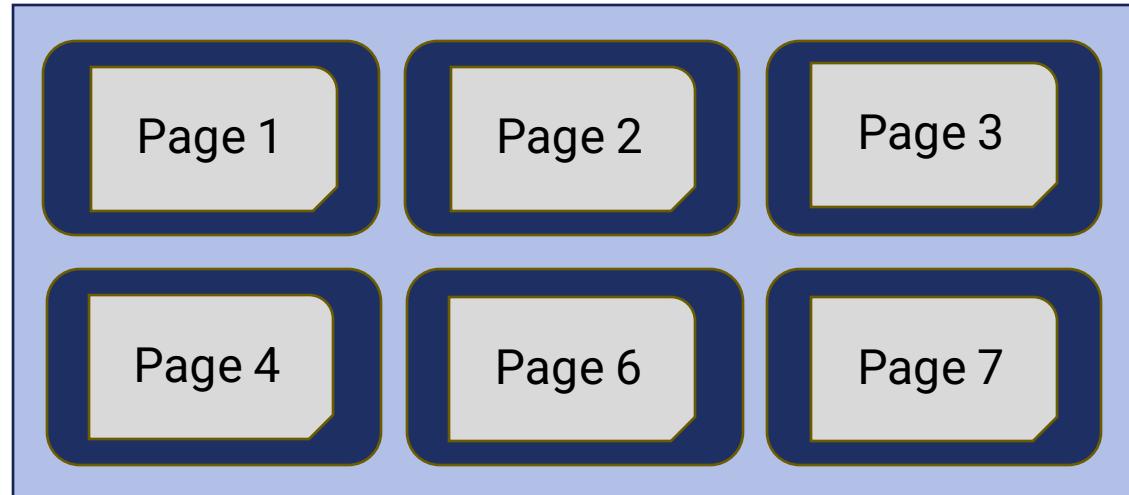
Disk Space Manager



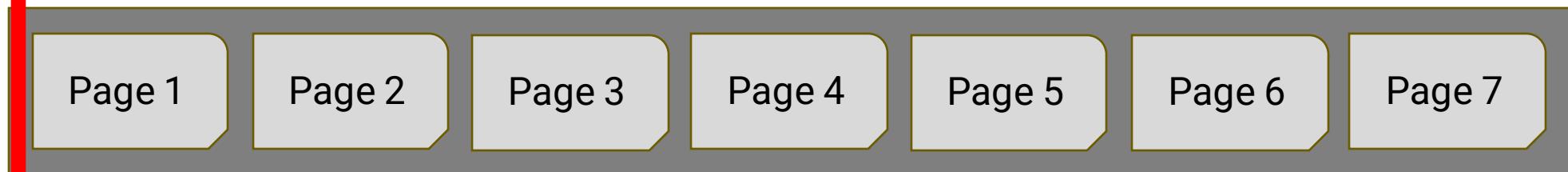
Repeated Scan (MRU): Read Page 1(againx2)

Cache Hit: 7

Attempts: 15



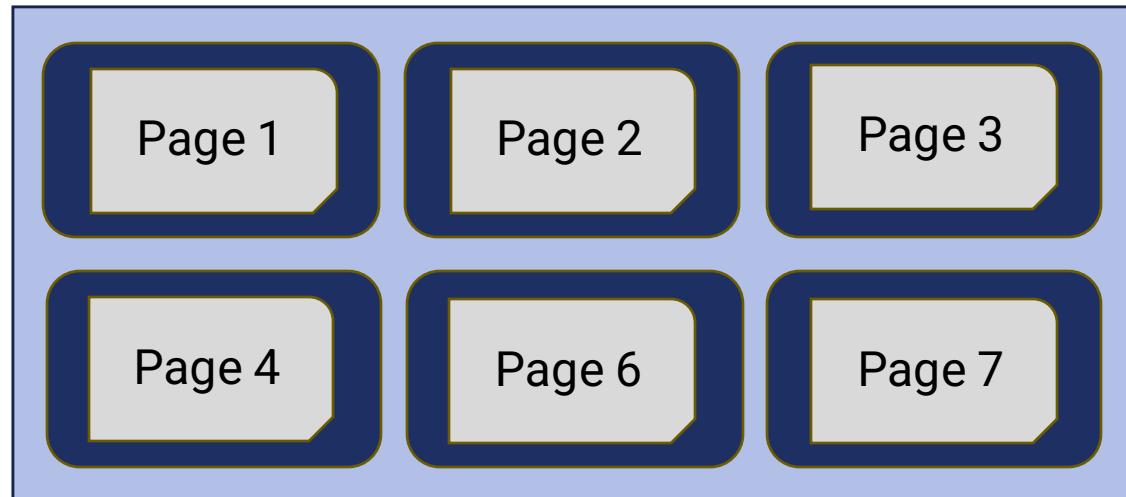
Disk Space Manager



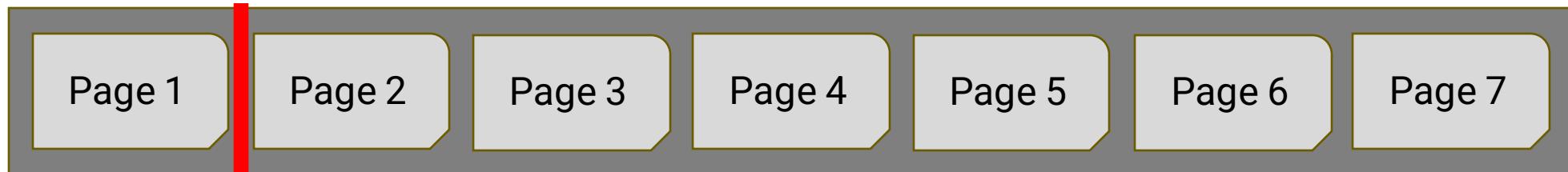
Repeated Scan (MRU): Read Page 2(againx2)

Cache Hit: 8

Attempts: 16



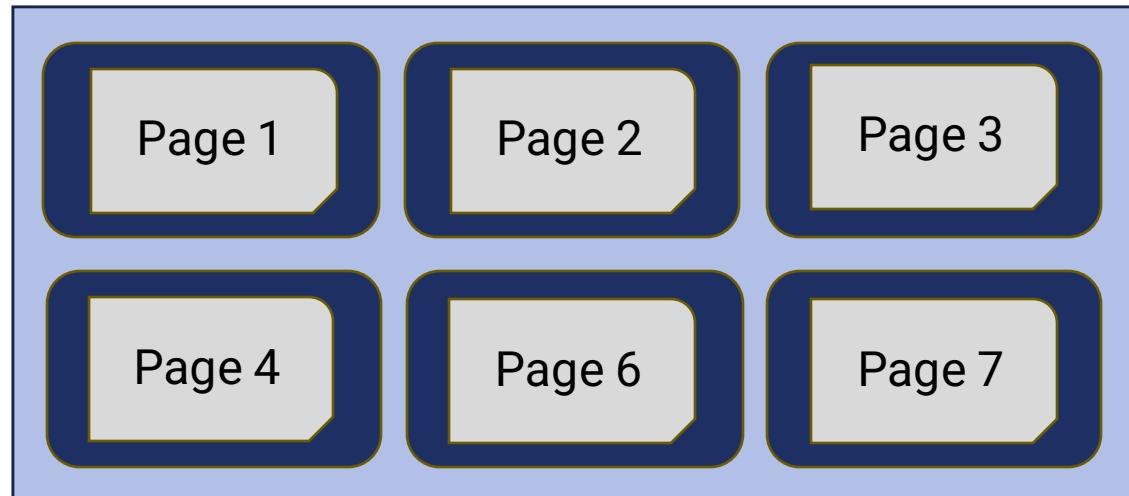
Disk Space Manager



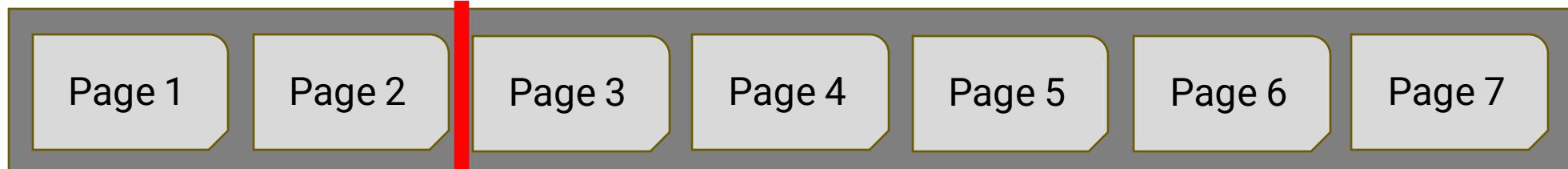
Repeated Scan (MRU): Read Page 3(againx2)

Cache Hit: 9

Attempts: 17



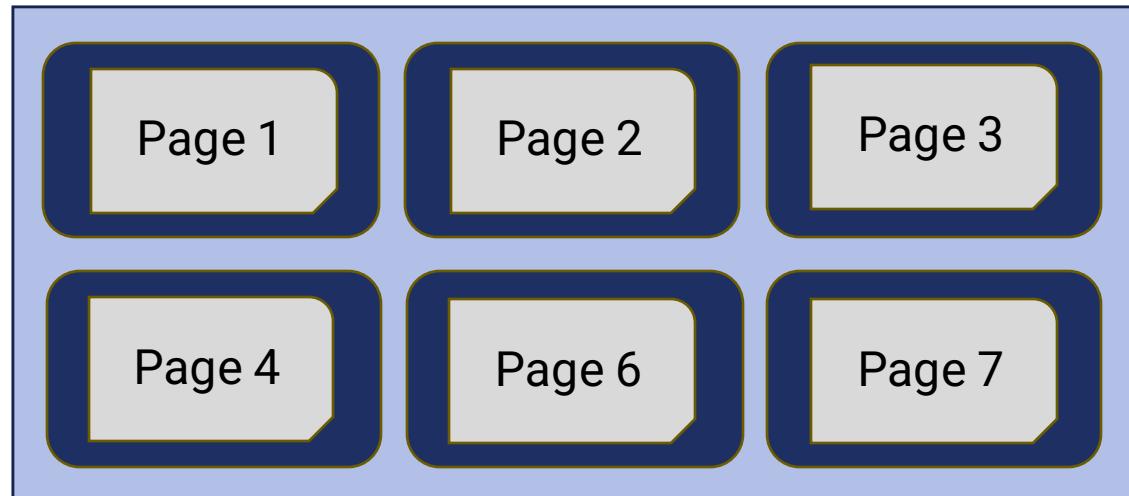
Disk Space Manager



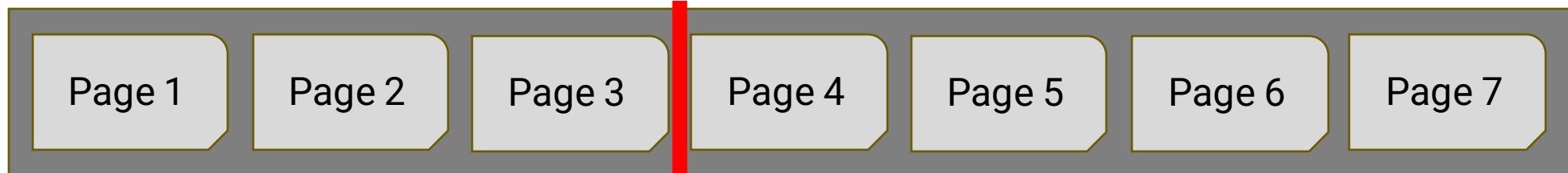
Repeated Scan (MRU): Read Page 4(againx2)

Cache Hit: 10

Attempts: 18



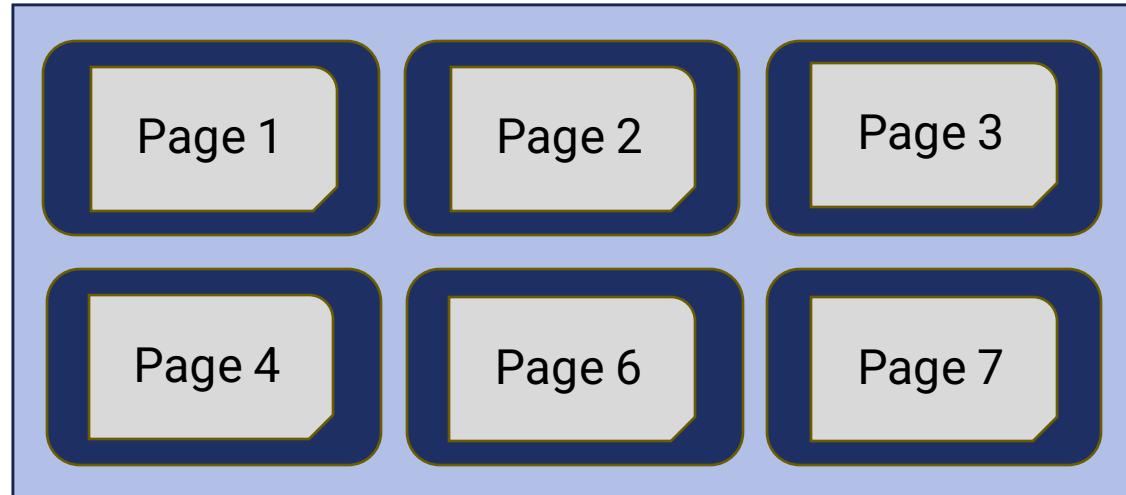
Disk Space Manager



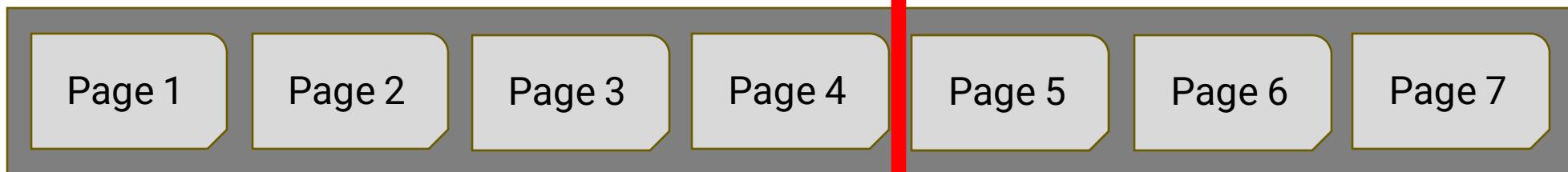
Repeated Scan (MRU): Read Page 5(againx2)

Cache Hit: 10

Attempts: 19



Disk Space Manager

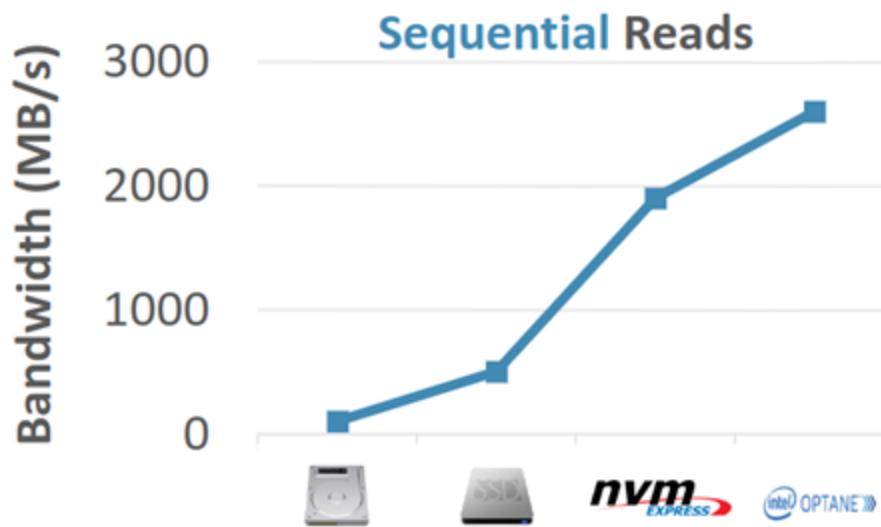


Sequential Flooding

- LRU: We need to get in/out every page
 - This is called **Sequential Flooding**
- MRU: performs the best in this case (**repeated scan**)

Again, *no replacement policy is guaranteed to be superior to the others.* The choice often depends on specific applications and their requirements.

New Trend: Disks are much faster



2010



2013



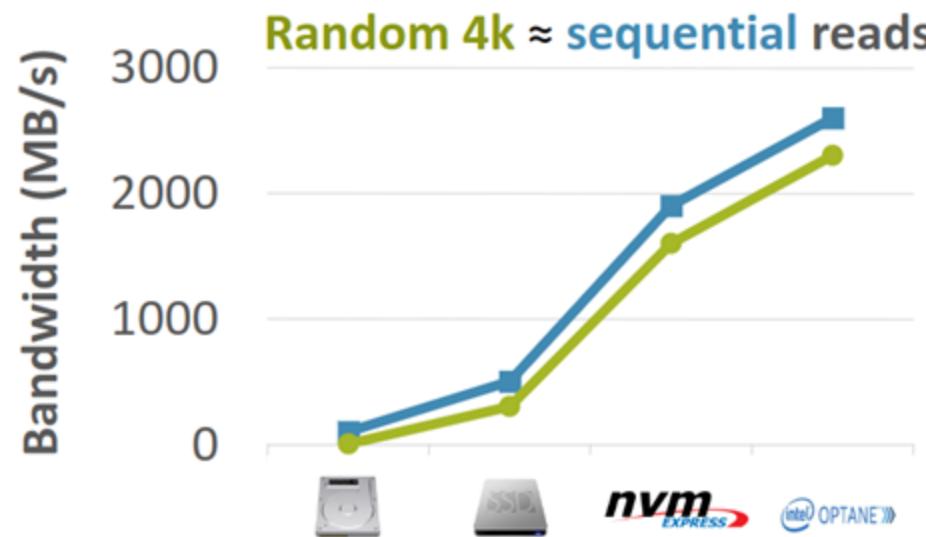
2016



2018



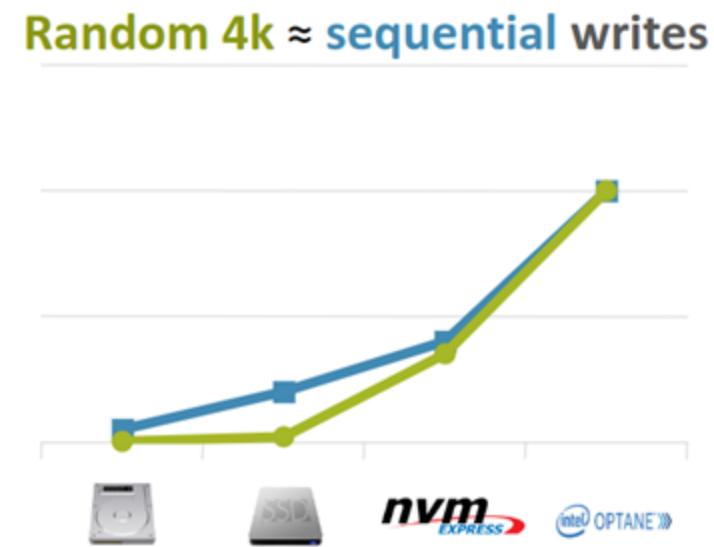
New Trend: Random vs Sequential Access



2010



2013



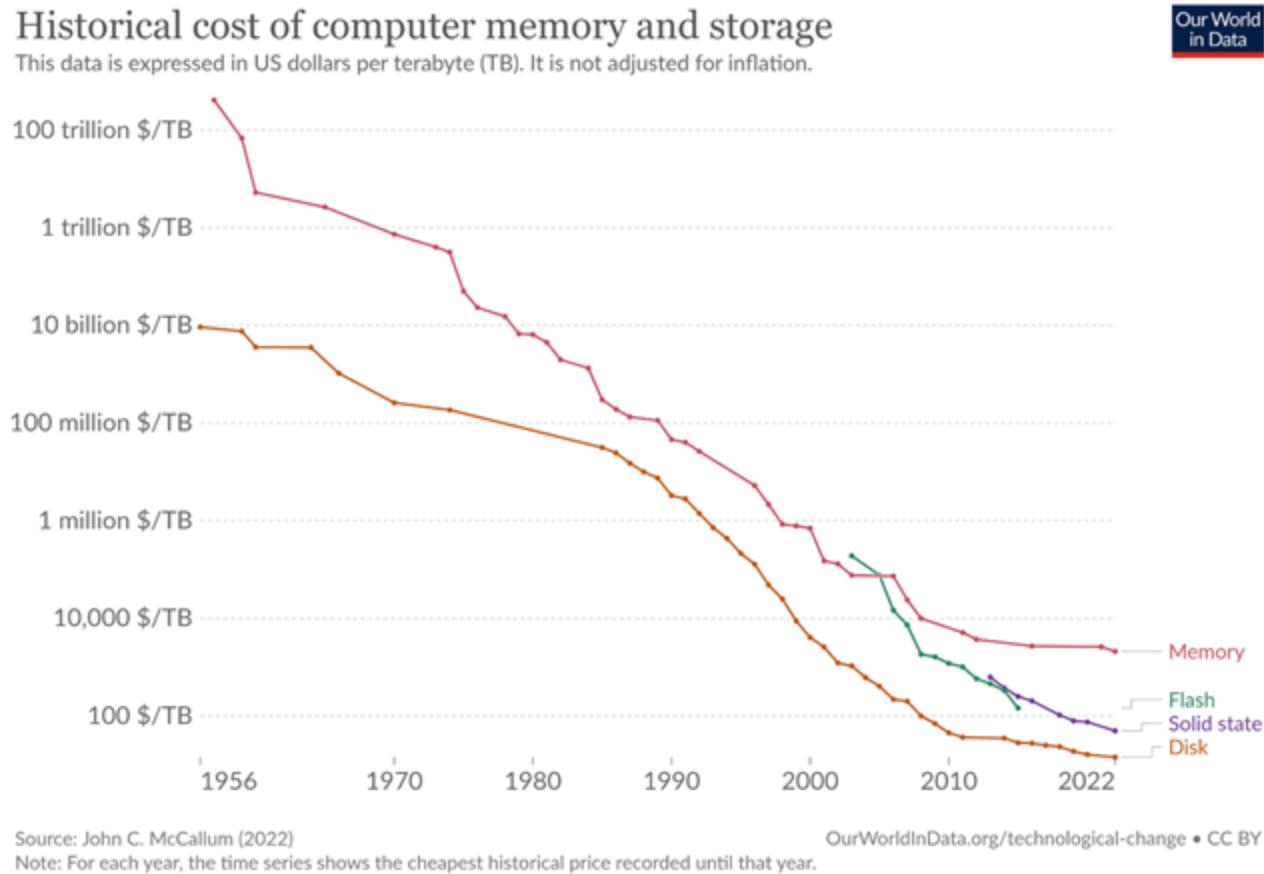
2016



2018



New Trend: Cheaper memory/disk



Overview: Block (Page) Formats

- **Block:** A block is a collection of *slots*.
- **Slot:** Each slot contains a record.
- **Record:** A record is identified by *record_id*: $\text{rid} = \langle \text{page id}, \text{slot number} \rangle$.

Question: How are records physically stored on disk?

Record Formats

Records are stored within fixed-length blocks.

- **Fixed-length:** each field has a fixed length as well as the number of fields.

33357462	Neil Young	Musician	0277
4 bytes	40 bytes	20 bytes	4 bytes

- Easy for intra-block space management.
- Possible waste of space.

- **Variable-length:** some field is of variable length.

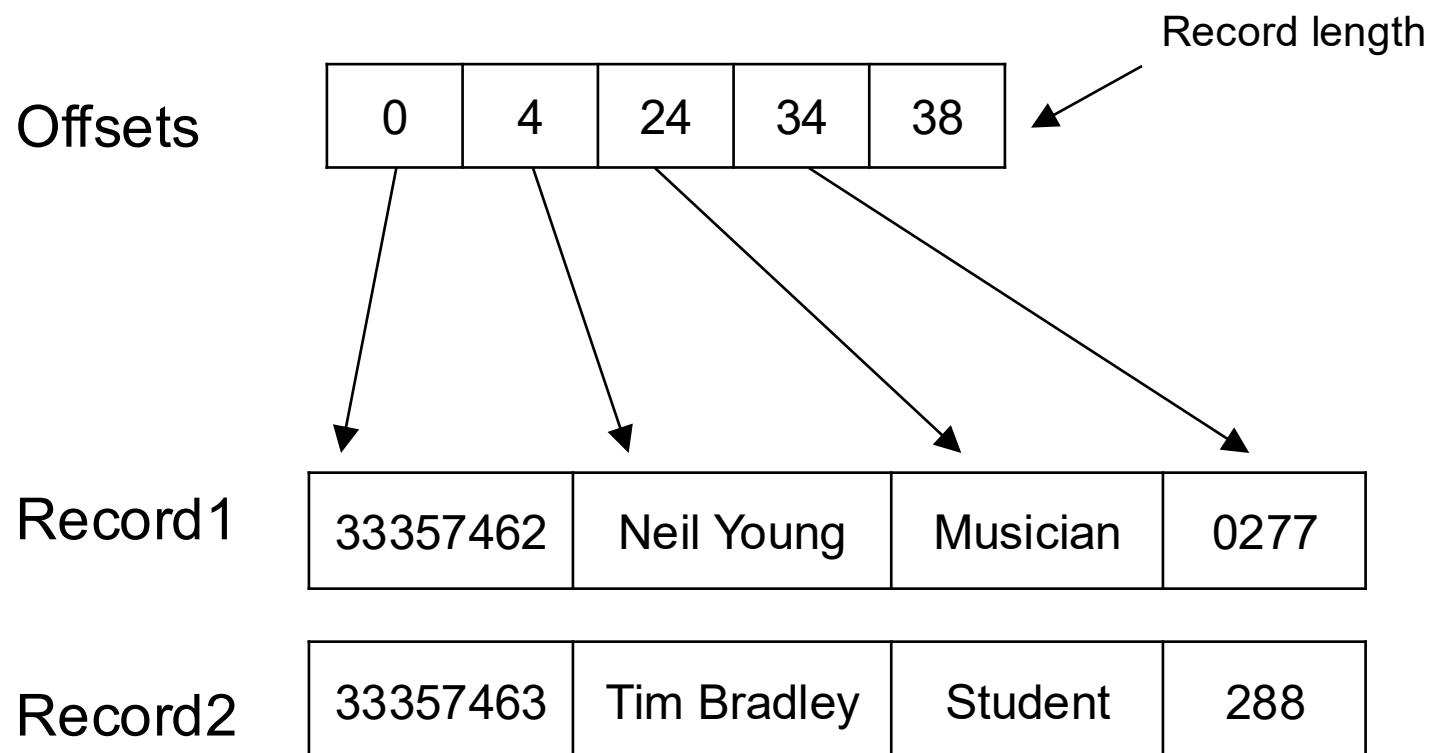
33357462	Neil Young	Musician	0277
4 bytes	10 bytes	8 bytes	4 bytes

- complicates intra-block space management
- does not waste (as much) space.

Fixed-Length

Encoding scheme for fixed-length records:

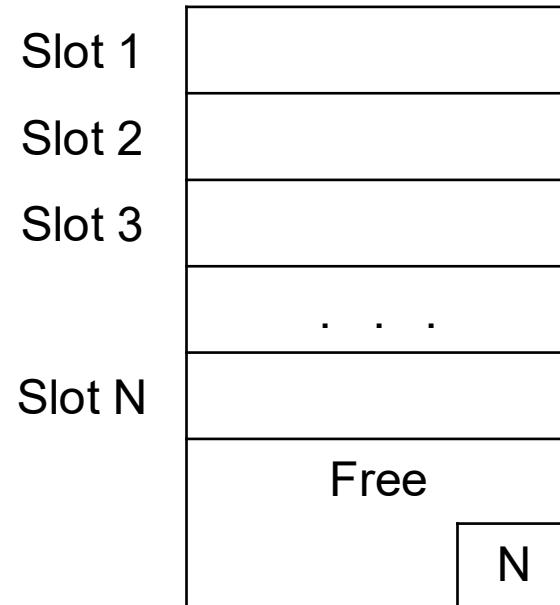
- length + offsets stored in header



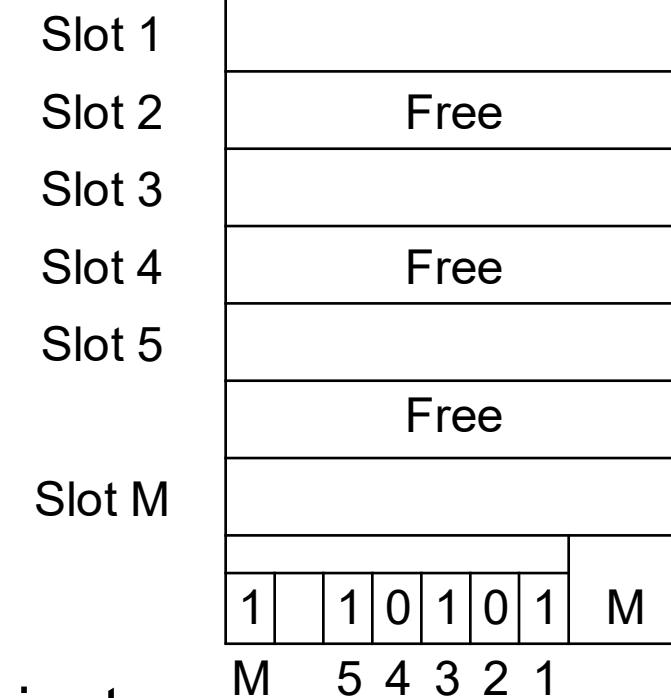
Fixed-Length Records

For fixed-length records, use record slots:

Packed



Unpacked, Bitmap



Insertion: occupy first free slot; packed more efficient.

Deletion: (a) need to compact, (b) mark with 0; unpacked more efficient.

Deletion in Packed Fixed-Length Records

Simple approach:

- Store record i starting from byte $n \times (i - 1)$, where n is the size of each record.

Consider two ways in
deleting record i :

- move records $i + 1, \dots, n$
to $i, \dots, n - 1$
- move record n to i

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Variable-Length

Encoding schemes where attributes are stored **in order**.

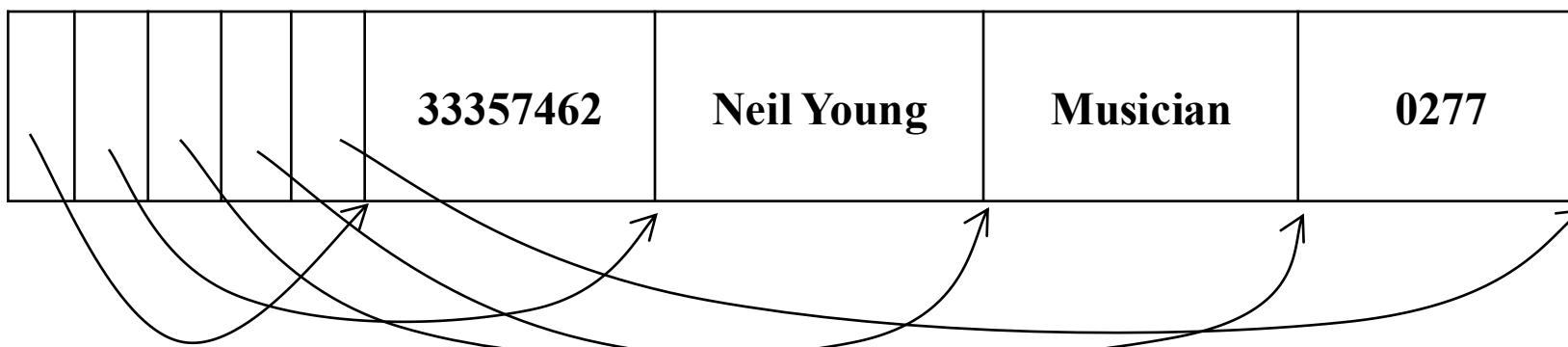
- Option 1: Prefix each field by length

4	xxxx	10	Neil Young	8	Musician	4	xxxx
---	------	----	------------	---	----------	---	------

- Option 2: Terminate fields by delimiter

33357462/Neil Young/Musician/0277/

- Option 3: Array of offsets



Variable-Length Records (1)

Another encoding scheme: attributes are not stored in order.

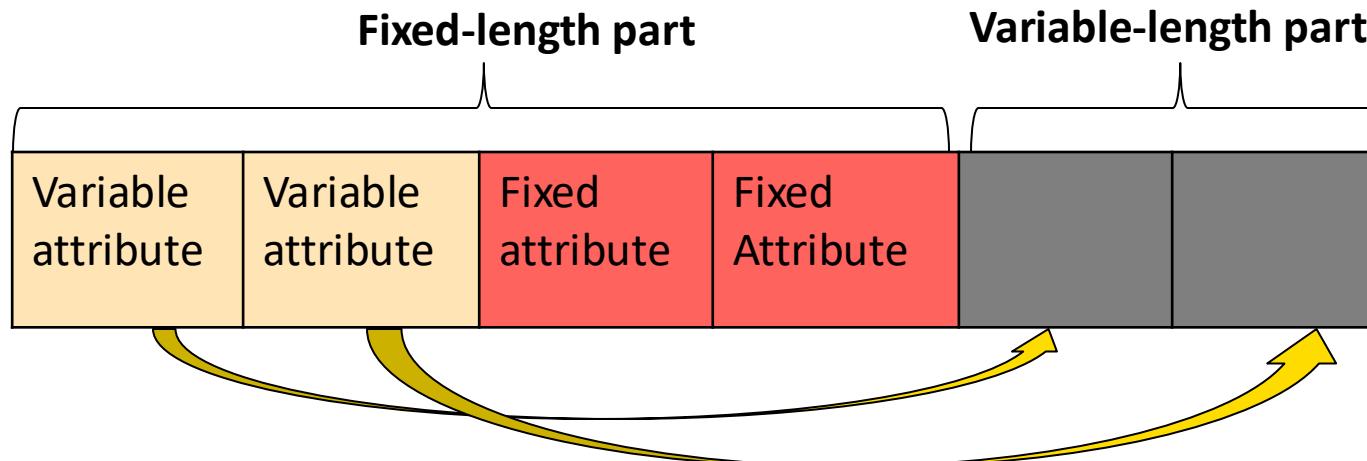
Fixed-length part followed by variable-length part.

- (b) The fixed-length part is to tell where we can find the data if it is a variable-length data field.
- (c) The variable-length part is to store the data.

Variable length attributes are represented by fixed size (**offset, length**) in the fixed-length part, and keep attribute values in the variable-length part.

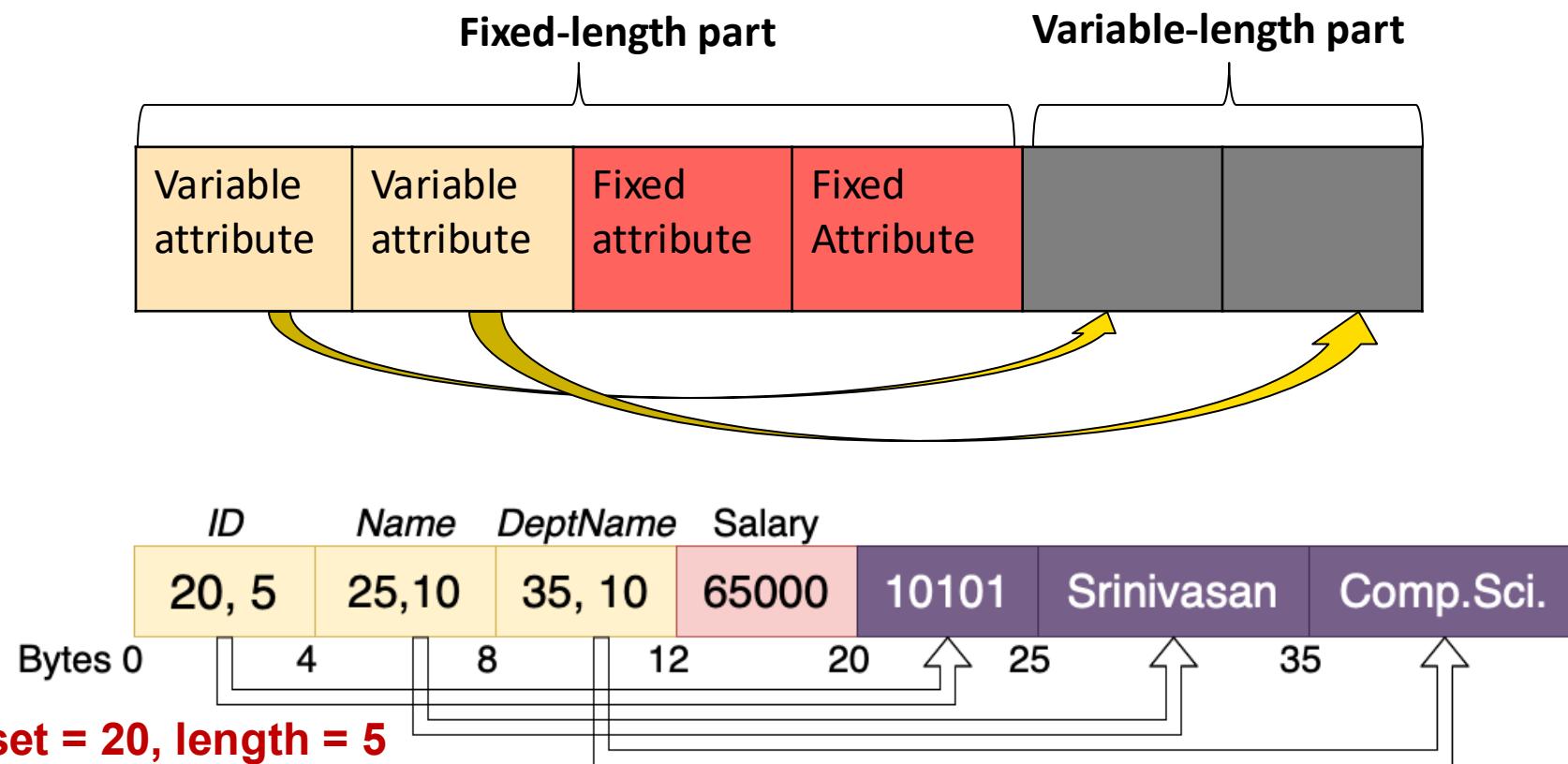
Fixed length attributes store attribute values in the fixed-length part.

Suppose there is a relation with 4 attributes: 2 fixed-length and 2 variable-length.



Variable-Length Records (2)

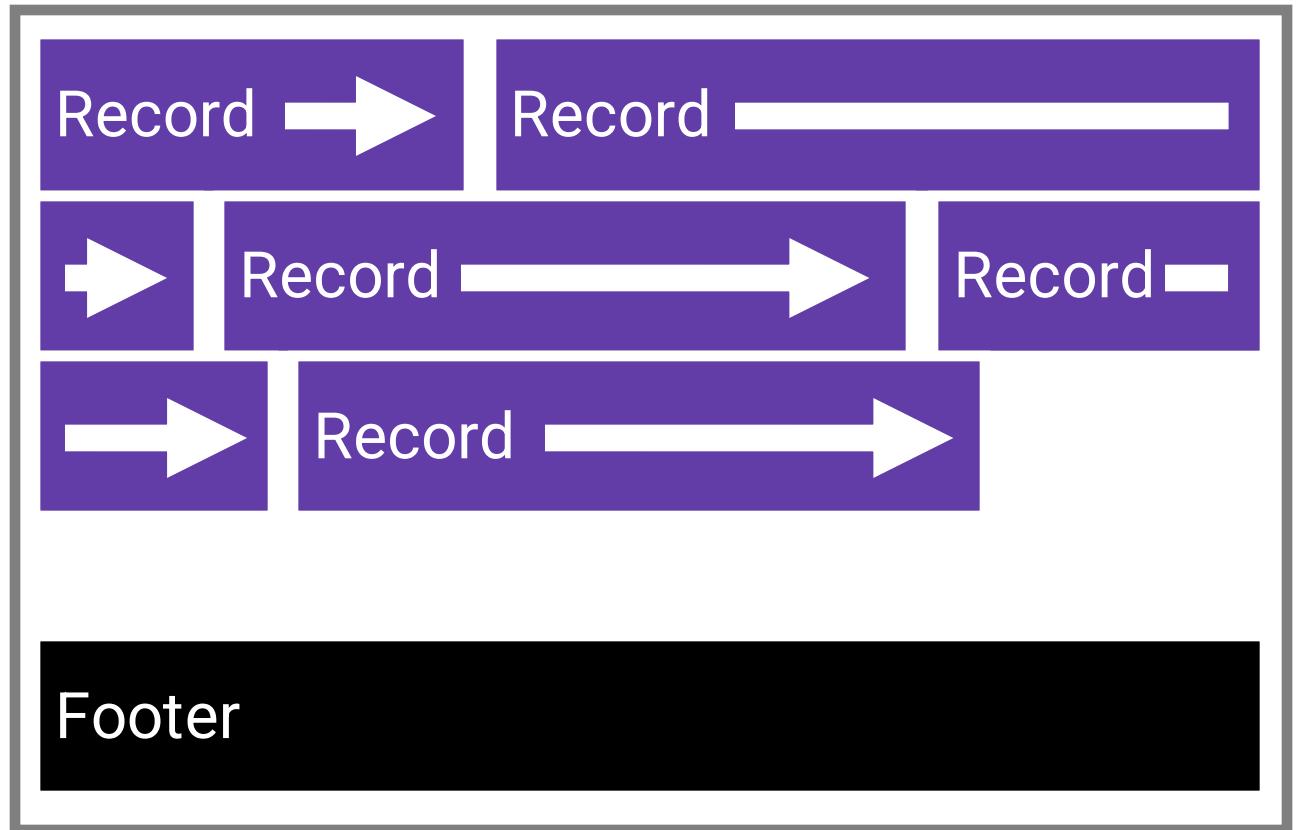
Example: a tuple of (**ID**, **Name**, **DeptName**, **Salary**) where the **first three** are variable length.



Variable-Length Records

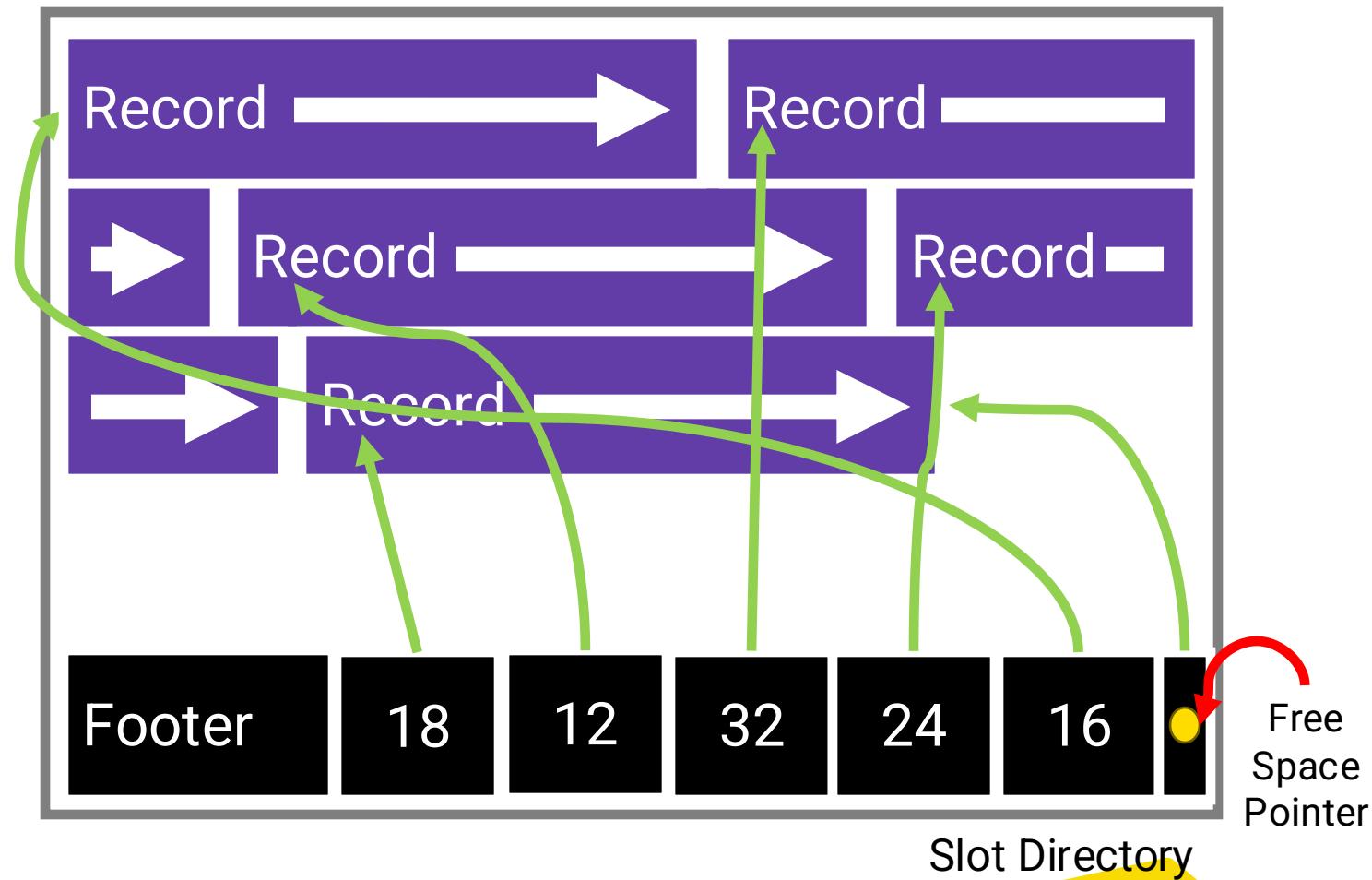
- How do we know where each record begins?
- What happens when we add and delete records?

Records metadata to footer



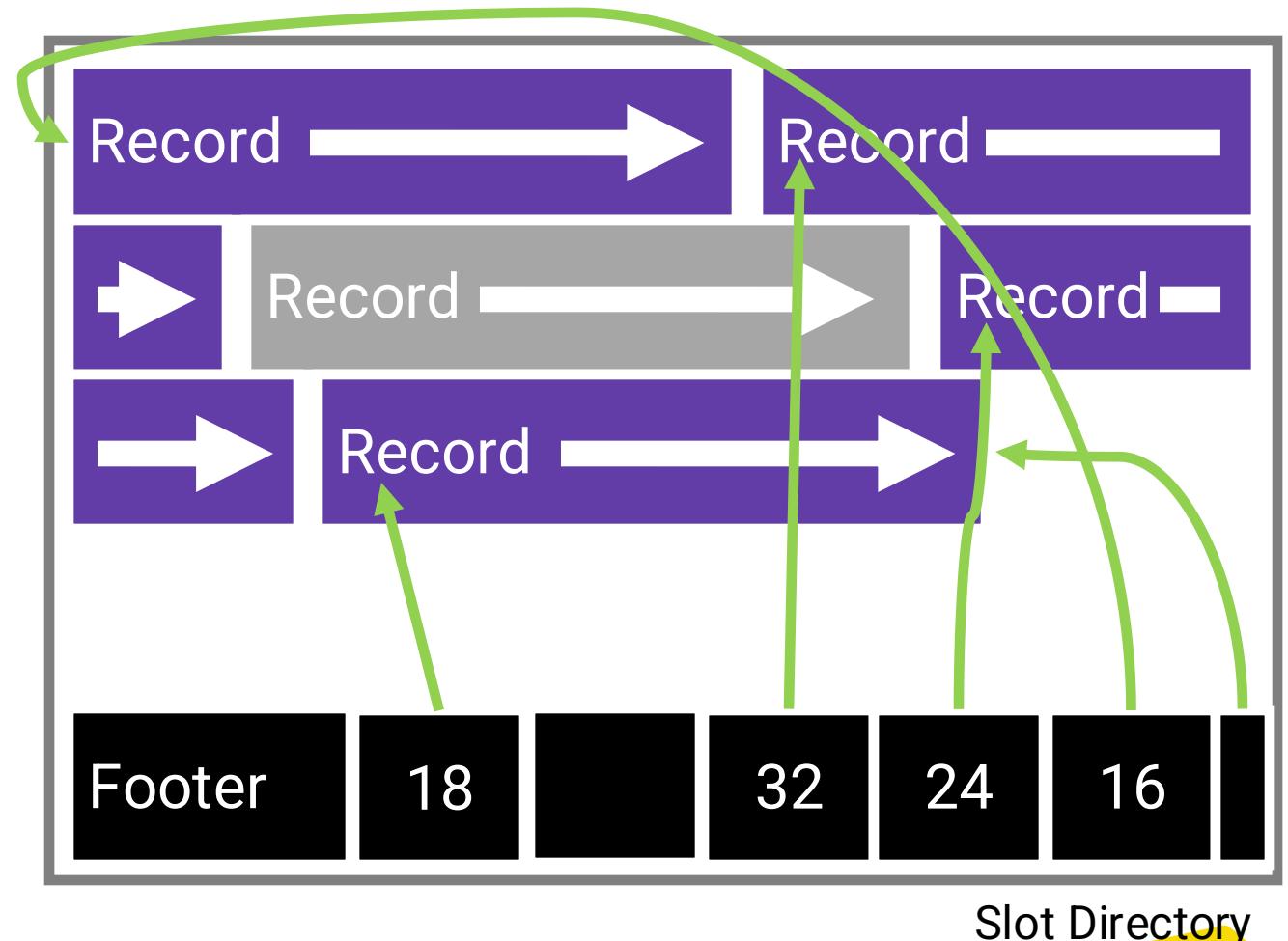
Slotted Page

- Introduce slot directory in footer
 - Pointer to free space
 - Length + Pointer to beginning of record
 - Reverse order
- Record ID = location in slot table
 - From right
- Delete?
 - E.g. 4th record on the page



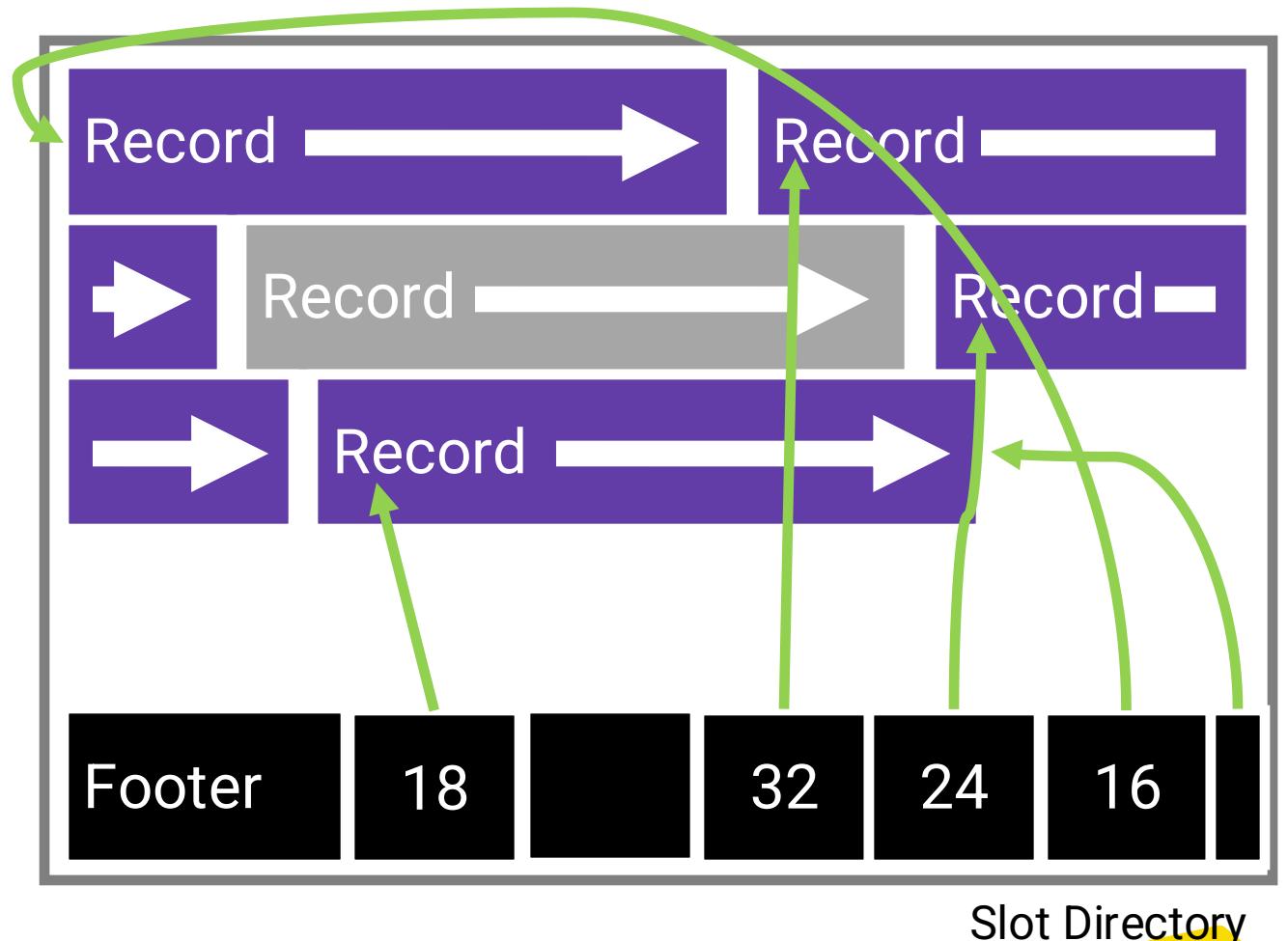
Slotted Page: Delete Record

- Delete record (Page 2, Record 4): Set 4th slot directory pointer to null
- Doesn't affect pointers to other records



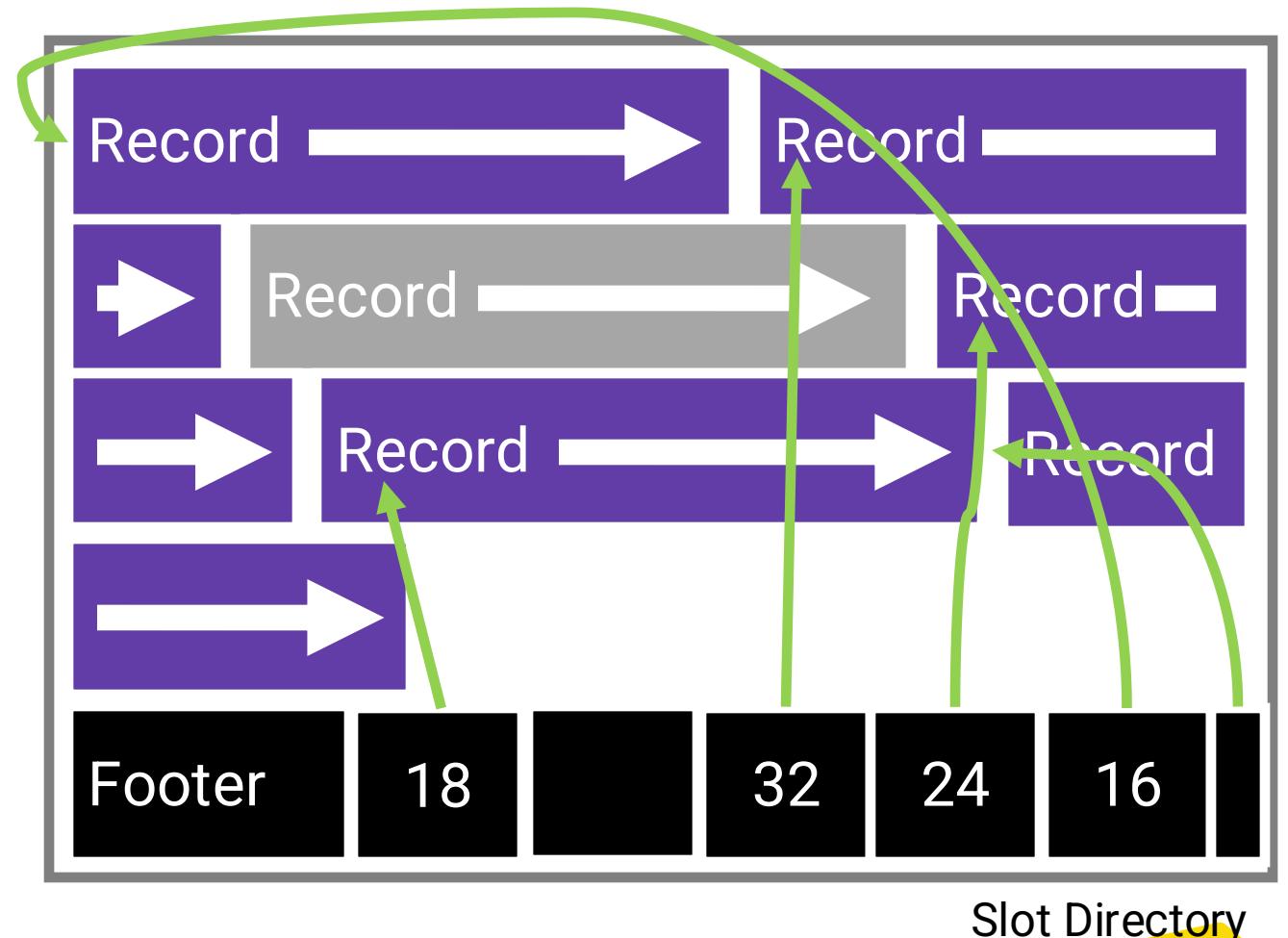
Slotted Page: Insert Record

- Insert:



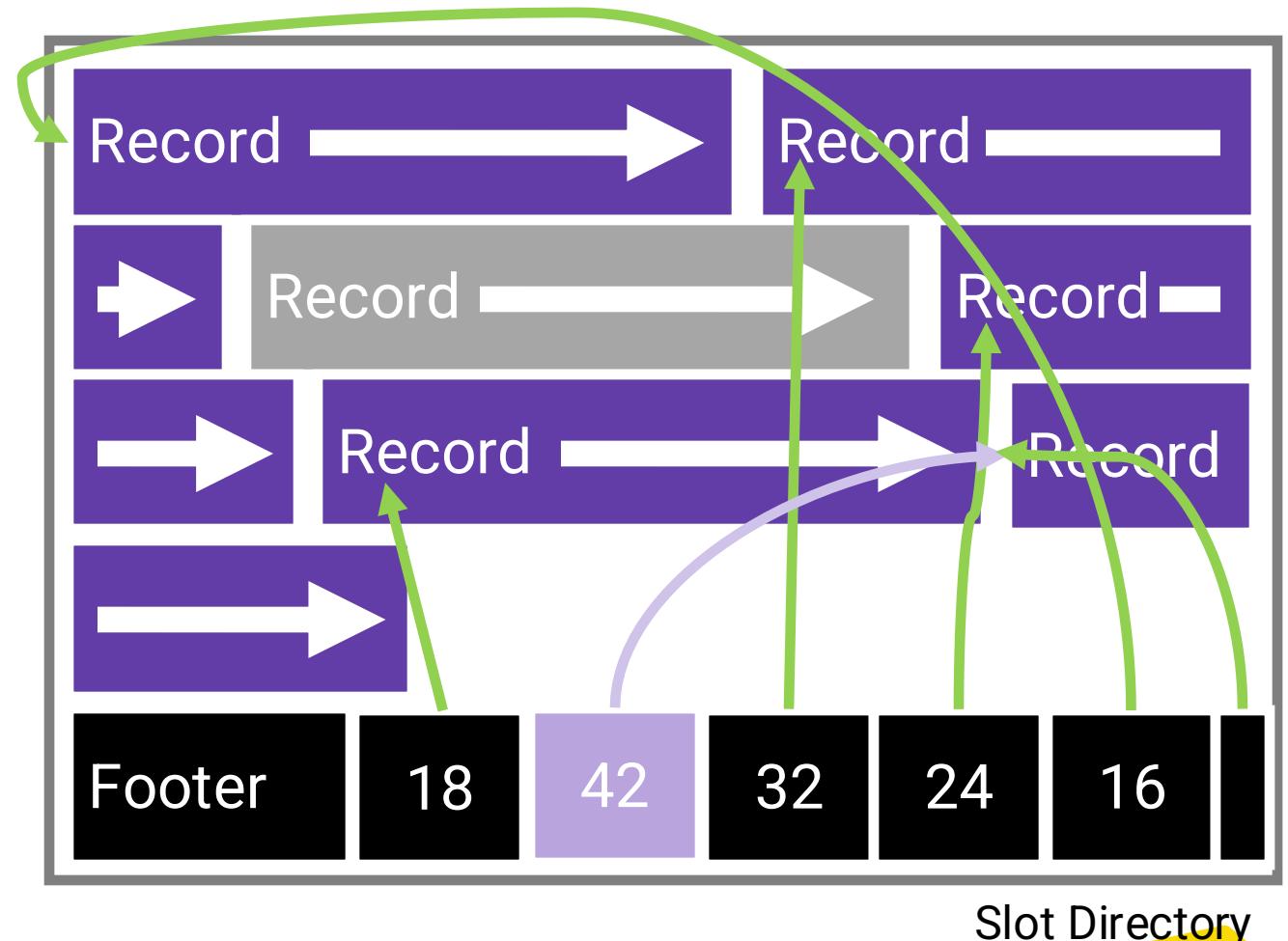
Slotted Page: Insert Record, Pt 2.

- Insert:
- Place record in free space on page



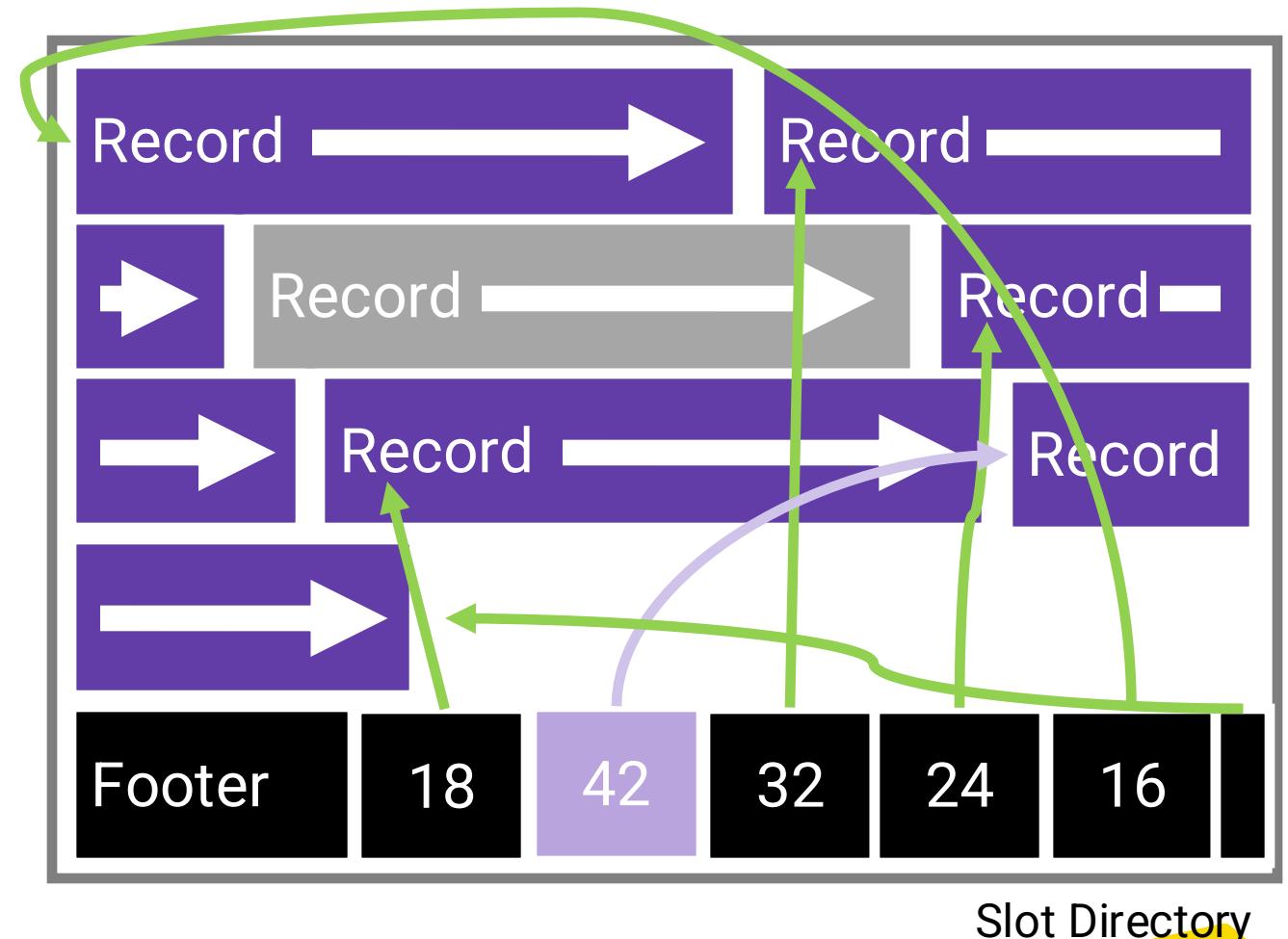
Slotted Page: Insert Record, Pt. 3

- Insert:
- Place record in free space on page
- Create pointer/length pair in next open slot in slot directory



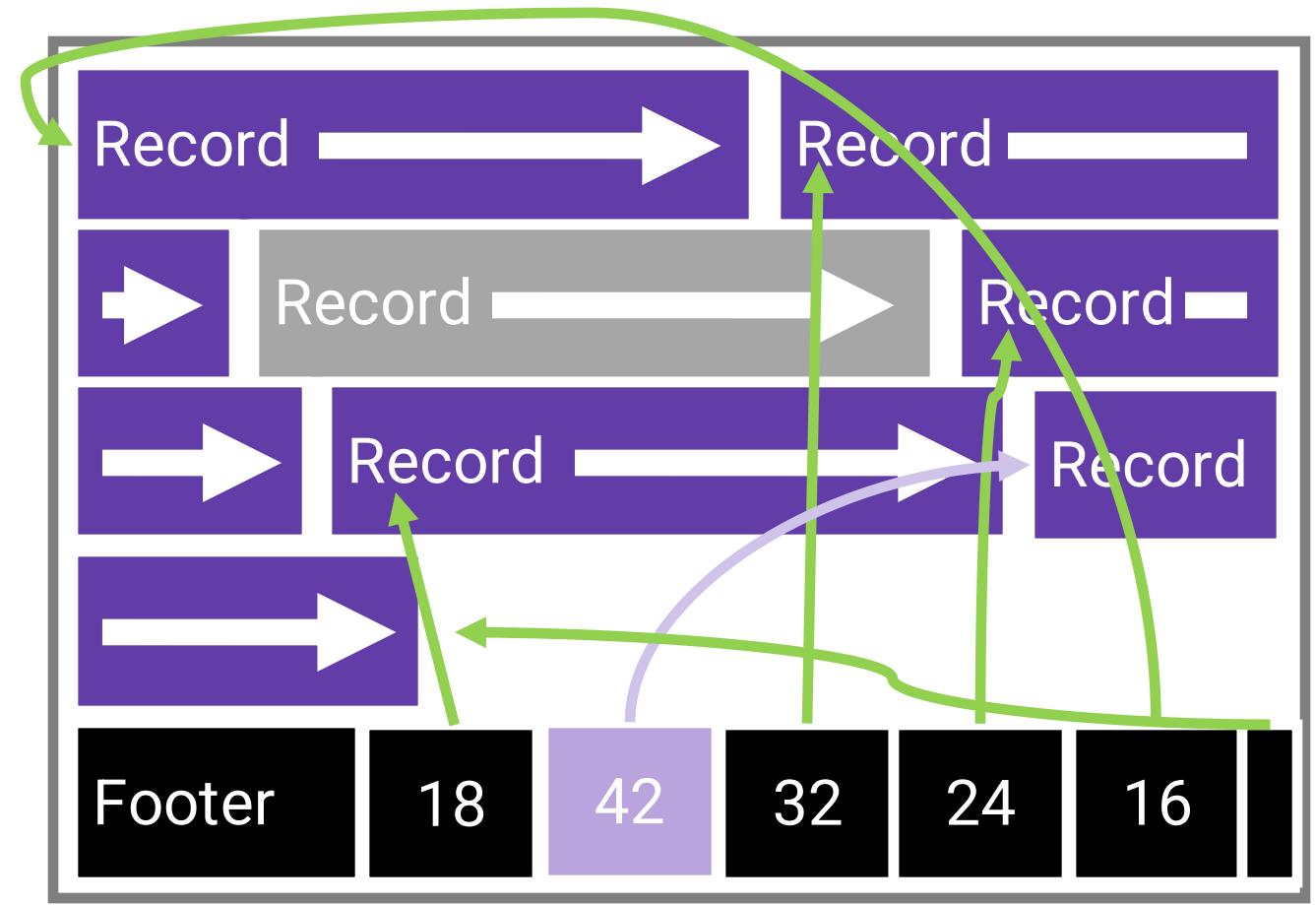
Slotted Page: Insert Record, Pt. 4

- Insert:
- Place record in free space on page
- Create pointer/length pair in next open slot in slot directory
- Update the free space pointer



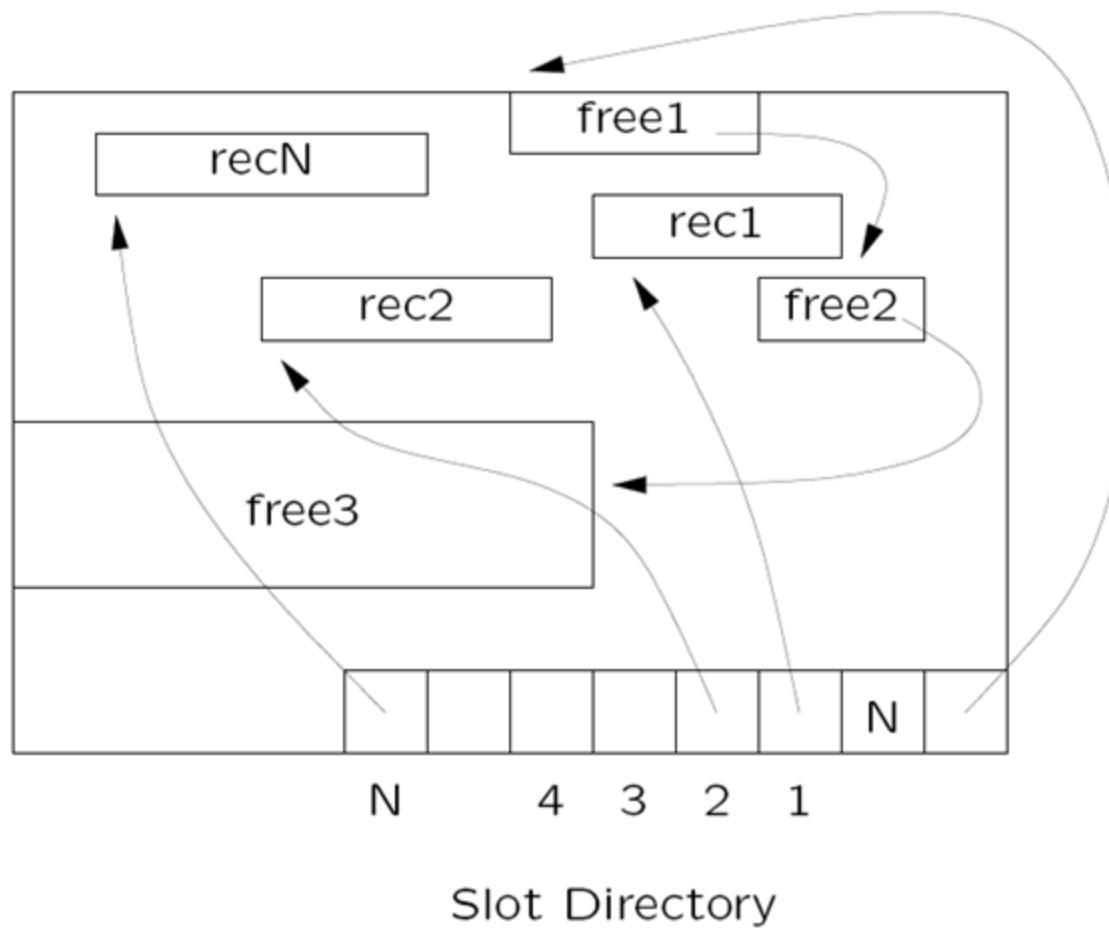
Slotted Page: Insert Record, Pt. 5

- Insert:
- Place record in free space on page
- Create pointer/length pair in next open slot in slot directory
- Update the free space pointer
- **Fragmentation?**



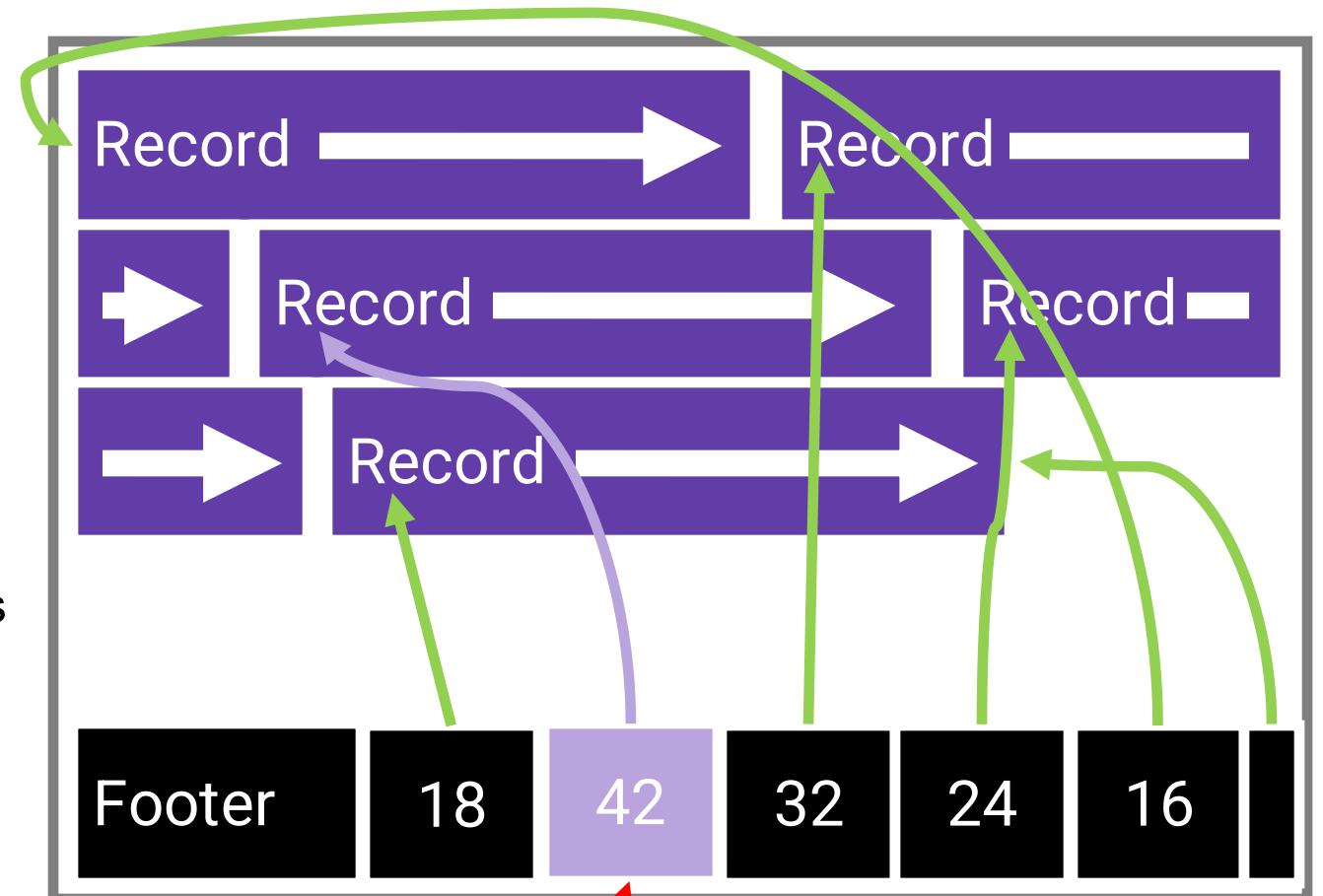
Variable-Length Records

Fragmented free space:



Fragmented Free Space

- Fragmentation?
 - Reorganise data on page!
- When should I reorganise?
 - We could reorganise on deletion
 - Too costly
 - Or wait until fragmentation blocks insertion
 - Modern RDBMS often do compaction when system is *idle*



Record ID:
(Page 2, Record 4)

Notes

Reminder:

- The basic store unit on disk (in memory) is block (page)
- We will use page/block interchangeably.
- One page consists of multiple data records.

Key Learning Outcomes

- Buffer replacement policies: how does each policy work
- Record / Page management

Next Week: Index, Transaction_Management