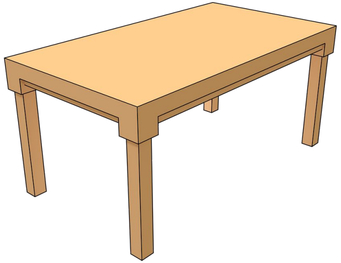
Databases and Big Data

Storage

Recap

- You have learned:
  - What a logical database looks like
- Also learned:
  - How to queries the database



```
db.student.find({"gpa": {"$gt": 3.5}})
```

```
select * from Student
where gpa > 3.5;
```

Next

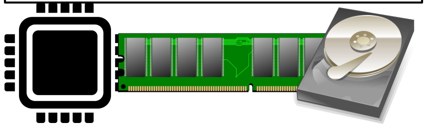
- How does the database answer queries
- How does it ***optimize*** them

Input:
SQL query

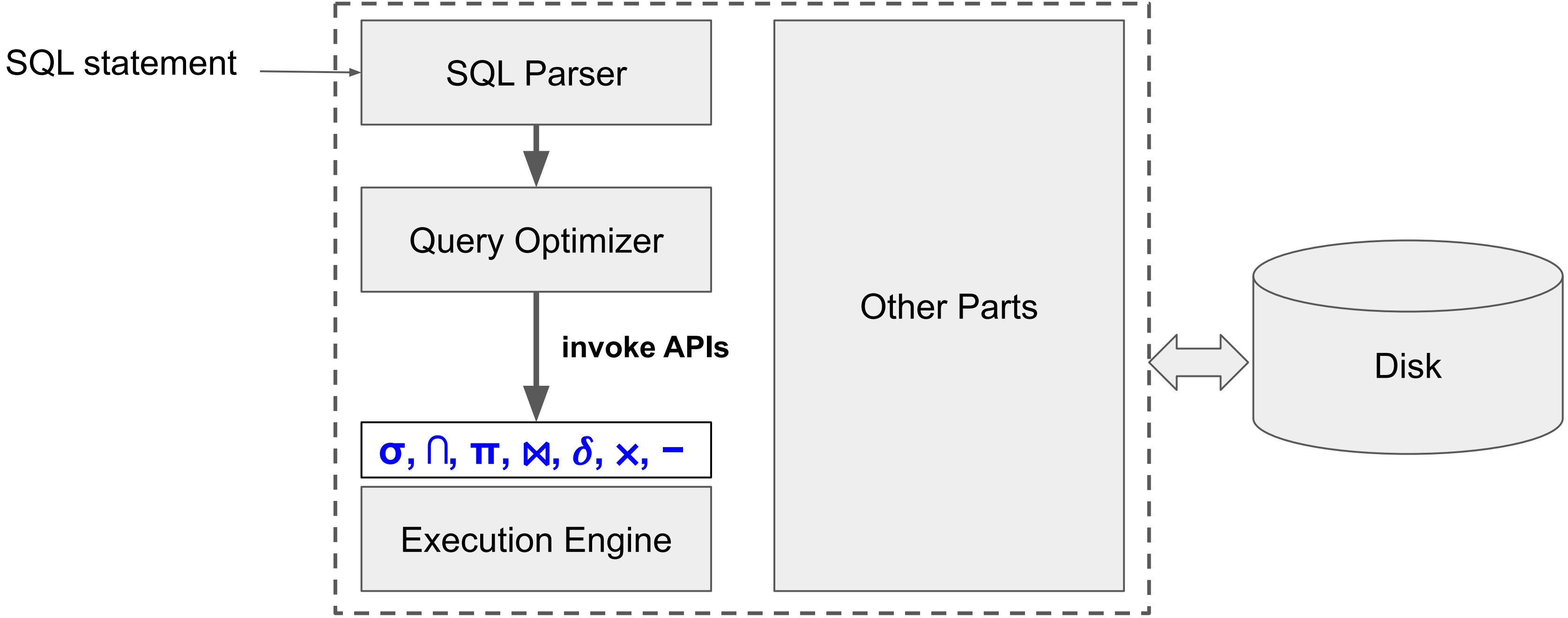
Optimizations

Data layout
Memory management
Data access
Query execution
etc.

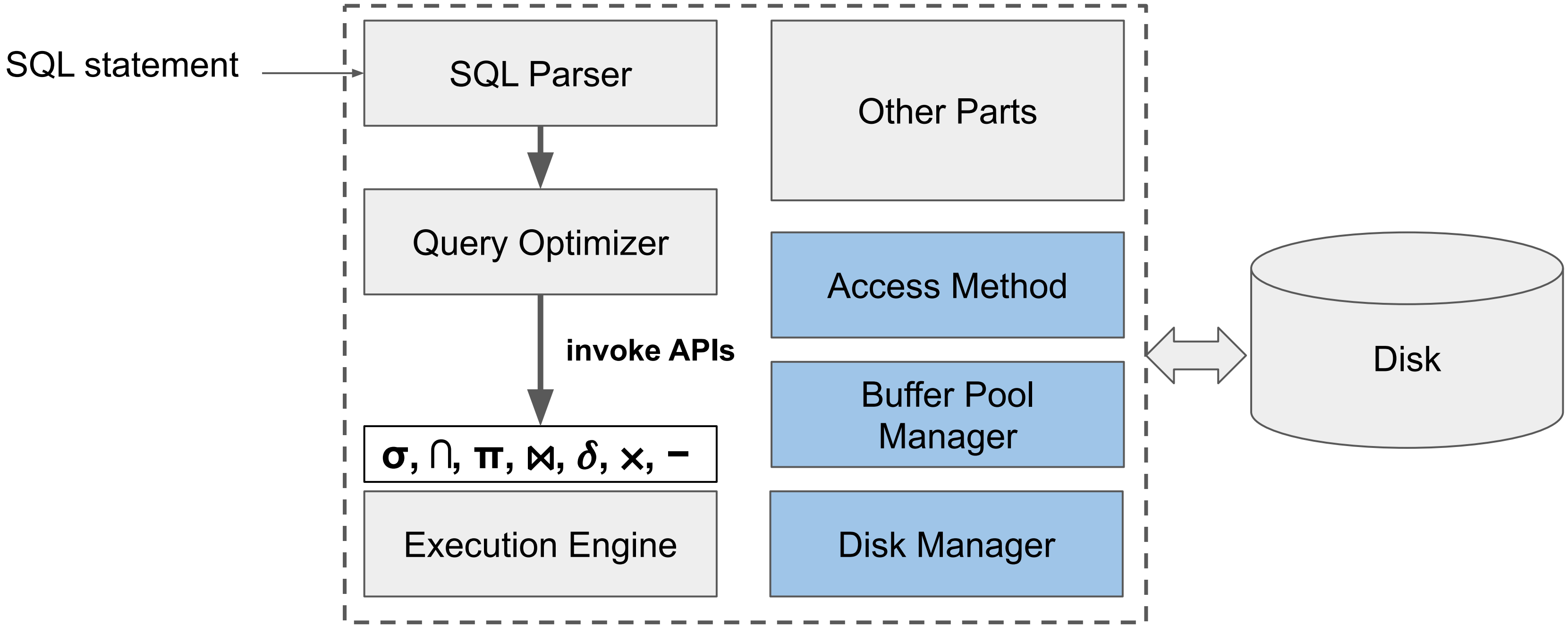
Output:
Data tuples



Glimpse Into Database Internal

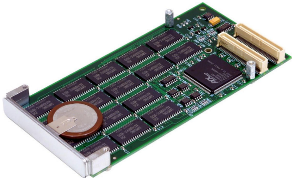


Database Internal



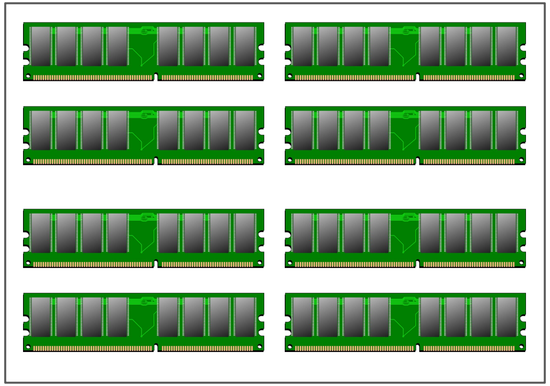
Our Scope

- Disk-based systems
  - Data stored on disk
  - Persistent, but slow
- Not in-memory systems
- Not SSD
- Not non-volatile memory

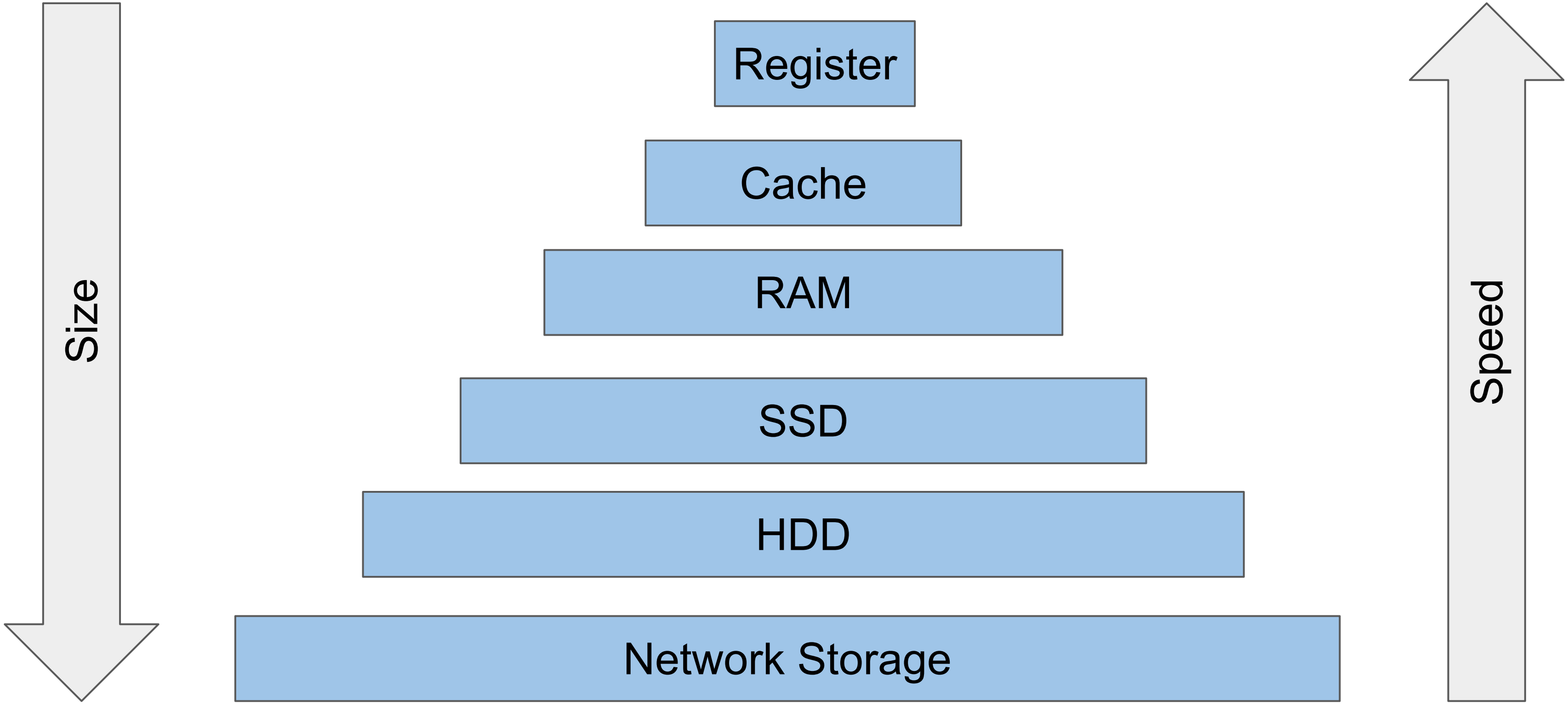


Storage Hierarchy

- A programmer's dream:
  - Private
  - Infinitely fast
  - Infinitely large
  - No failure
  - Cheap



Storage Hierarchy

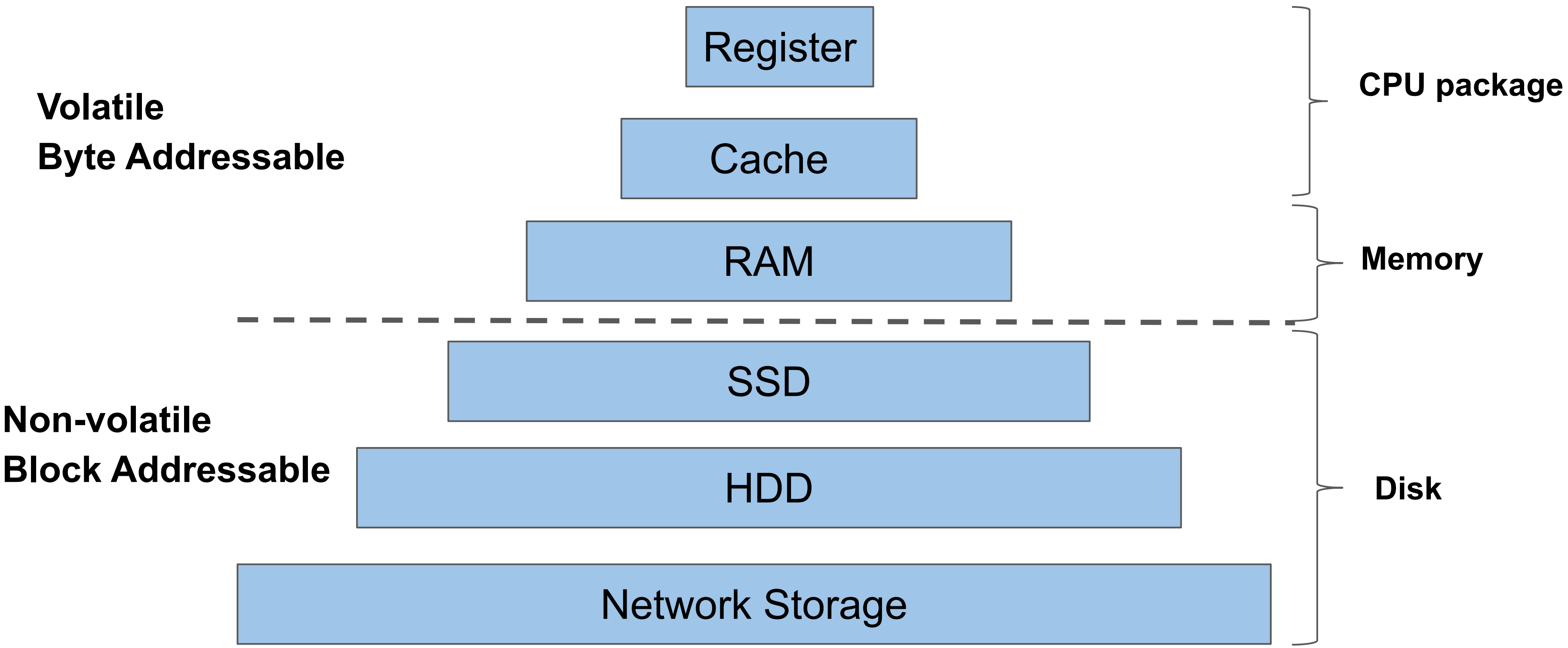


Latency Numbers Every Programmer Should Know

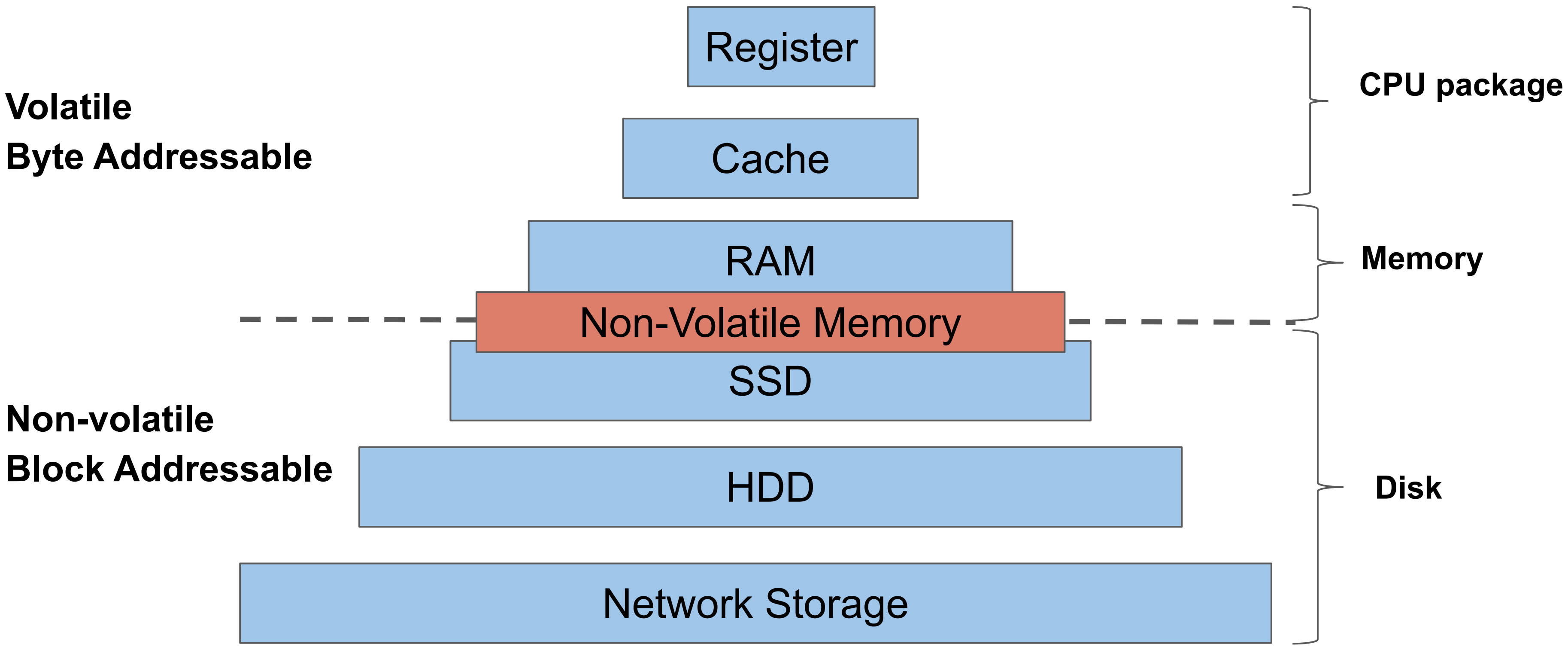
```
L1 cache reference ......................... 0.5 ns
Branch mispredict ............................ 5 ns
L2 cache reference ........................... 7 ns
Mutex lock/unlock ........................... 25 ns
Main memory reference ...................... 100 ns
Compress 1K bytes with Zippy ............. 3,000 ns  =   3 µs
Send 2K bytes over 1 Gbps network ....... 20,000 ns  =  20 µs
SSD random read ........................ 150,000 ns  = 150 µs
Read 1 MB sequentially from memory ..... 250,000 ns  = 250 µs
Round trip within same datacenter ...... 500,000 ns  = 0.5 ms
Read 1 MB sequentially from SSD* ..... 1,000,000 ns  =   1 ms
Disk seek ........................... 10,000,000 ns  =  10 ms
Read 1 MB sequentially from disk .... 20,000,000 ns  =  20 ms
Send packet CA->Netherlands->CA .... 150,000,000 ns  = 150 ms
```

[Jeff Dean]

Storage Hierarchy



Storage Hierarchy



System Design Goal

- Memory management:
  - Clever use of memory to avoid expensive disk operations
- Important insight:
  - Sequential access are much, much better than random access

Can we just use Virtual Memory?

t_s: seek time ~ 20ms
t_r: rotate time ~ 10ms
t_{tf}: transfer time ~ 0.1ms
D: # blocks

Random access time:

$$T_1 = D(t_s + t_r + t_{tf})$$

Sequential access time:

$$T_2 = t_s + t_r + Dt_{tf}$$

$D = 1000$
$T_1 \sim 100T_2$

Disk Manager

- Problem:
  - How to lay out data on disk
- Approach:
  - Store data (tables) in multiple files
  - Leverage the OS's file system

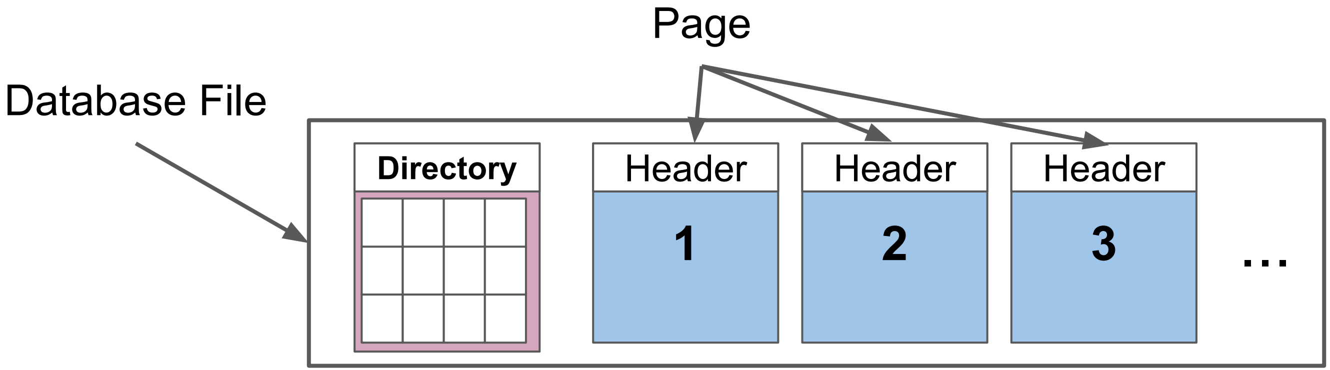
Disk Manager

- Problem:
  - How to lay out data on disk
- Approach:
  - Store data (tables) in multiple files
  - Leverage the OS's file system

But didn't we learn there're many problems with file systems, in Lecture 1?

Disk Manager

- Database stored as a collection of files
- Each file is organized as a collection of pages
  - Don’t believe me? See it for yourself `hexdump -C <filename>`

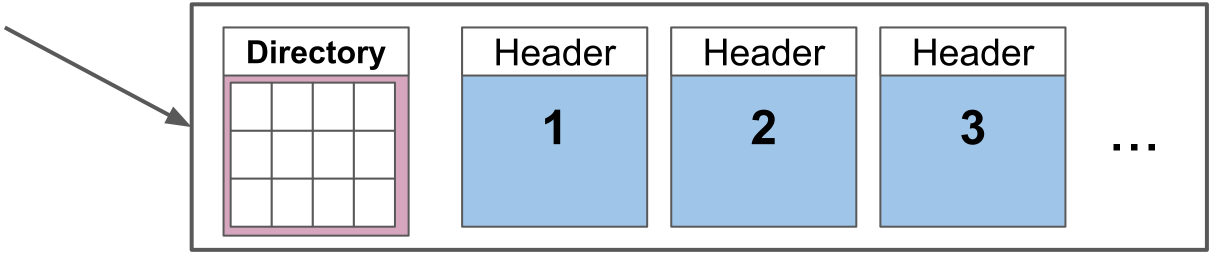


Disk Manager

- Database file vs. Linux file:
  - Page vs. block
  - Directory vs. inode
- How about “page”?

Where?	Page Size
Hardware page	4K
OS page	4K
DB page	1-16K

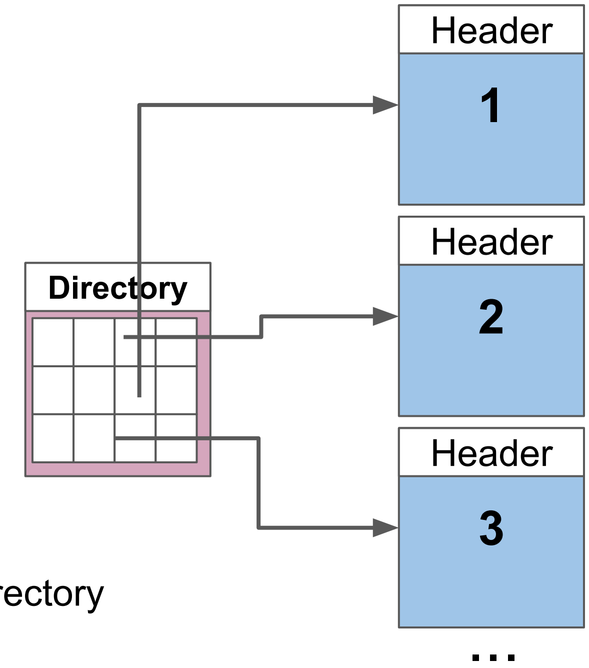
Database File



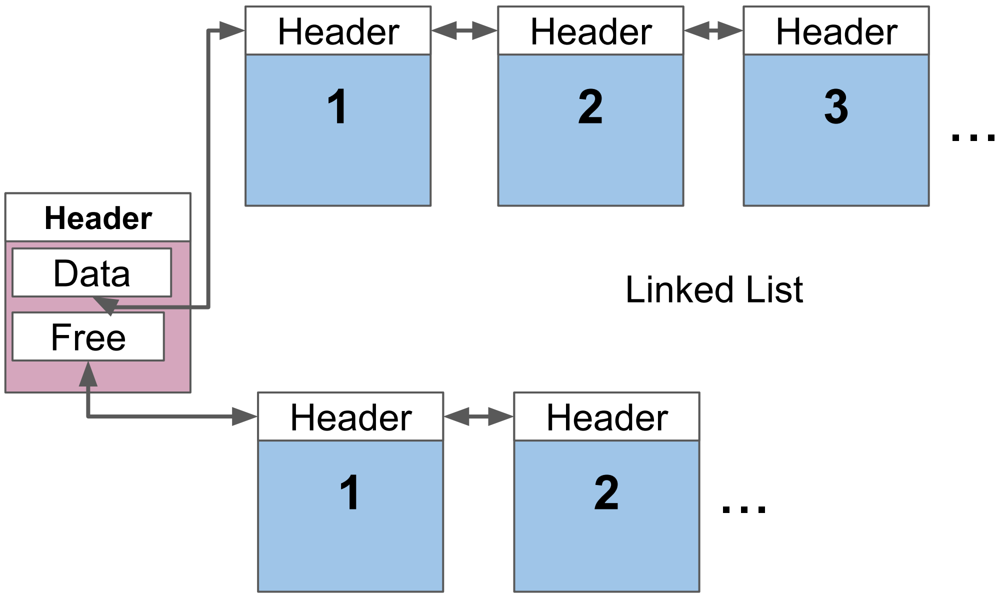
Disk Manager

- Problem 1: How to organize pages
  - **Heap file:** *unordered* collection of pages

Not to be confused with Heap data structure



Directory



Linked List

Disk Manager

- Cost of heap file
  - Assume directory pages fit in memory

Pros:
- Good for bulk insertion.
- For a small database, fetching of records is faster than the sequential record.

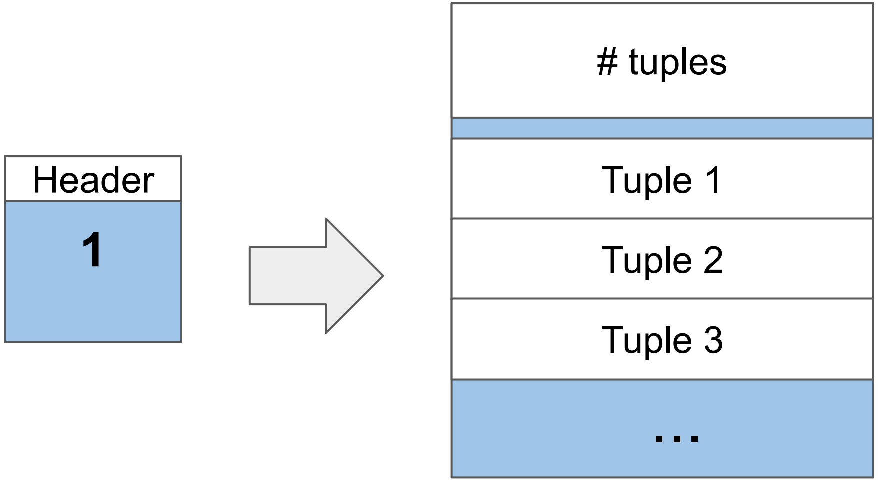
Cons:
- Inefficient for large database cause it takes time to search/ modify record.

	Sequential layout (BEST!)	Random layout (WORST!)
Insert	$t_{s+r} + 2.t_{tf}$	$t_{s+r} + 2.t_{tf}$
Lookup	$t_{s+r} + t_{tf}.D/2$	$(t_{s+r} + t_{tf}).D/2$
Scan	$t_{s+r} + t_{tf}.D$	$(t_{s+r} + t_{tf}).D$

every page you wanna look need to look again

Disk Manager

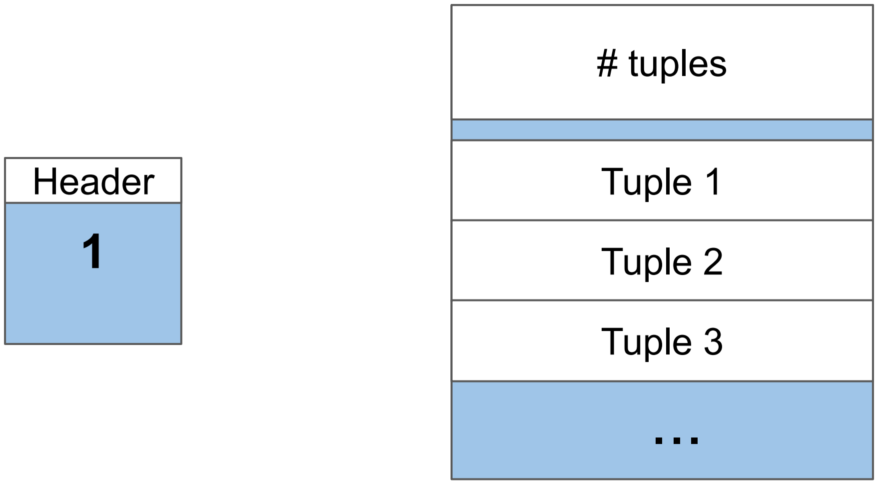
- Problem 2: How to organize data in a page?



	ID	Name	Job	Salary
tuple 1				
tuple 2				
tuple 3				

Disk Manager

- Problem 2: How to organize data in a page?



NOT great!

Why?

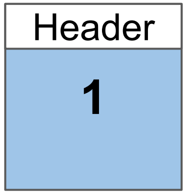
- Tuple may not be of same size
if we allocate fixed space for tuple -> waste space

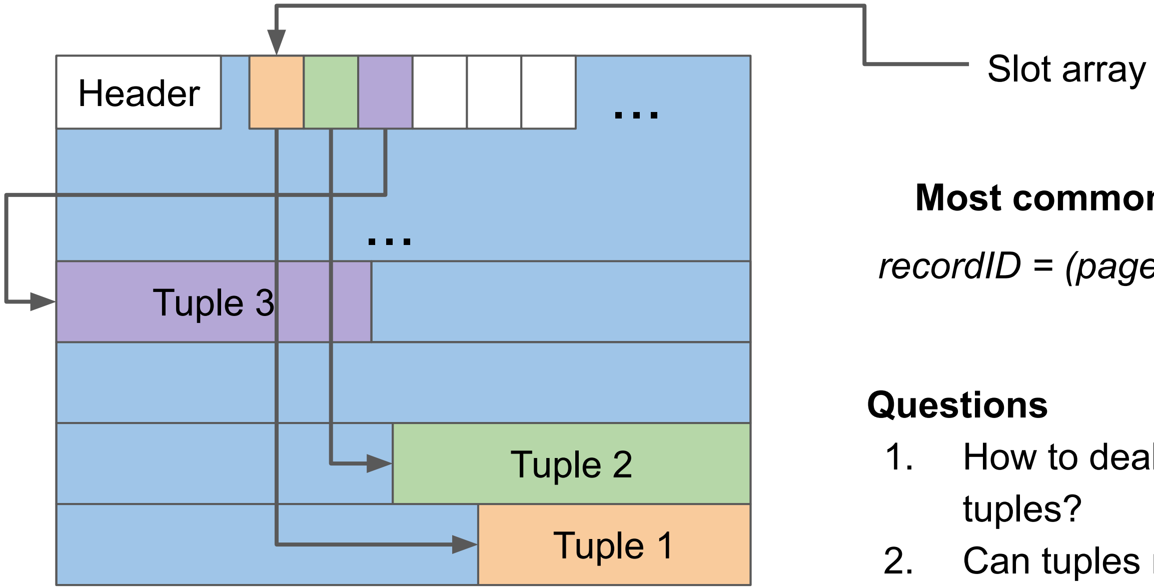
- When we delete things (if tuples not fixed size), we get fragmentations

Disk Manager

Another structure here (small structure) that contains pointers to the beginning address and ending address of the tuple.

- Problem 2: How to organize data in a page?





Most common layout

recordID = (pageID, slotID)

Questions

1. How to deal with variable-length tuples?
2. Can tuples move around?

Disk Manager

- Problem 3: What's in the tuple?

Salary values are usually similar (e.g. 1000, 2000, 3000 etc) so it is more efficient to store them together.
- Save space

ID	Name	Job	Salary

ID	Name	Job	Salary

MySQL

Row Store

+ Fast update, insert, etc.

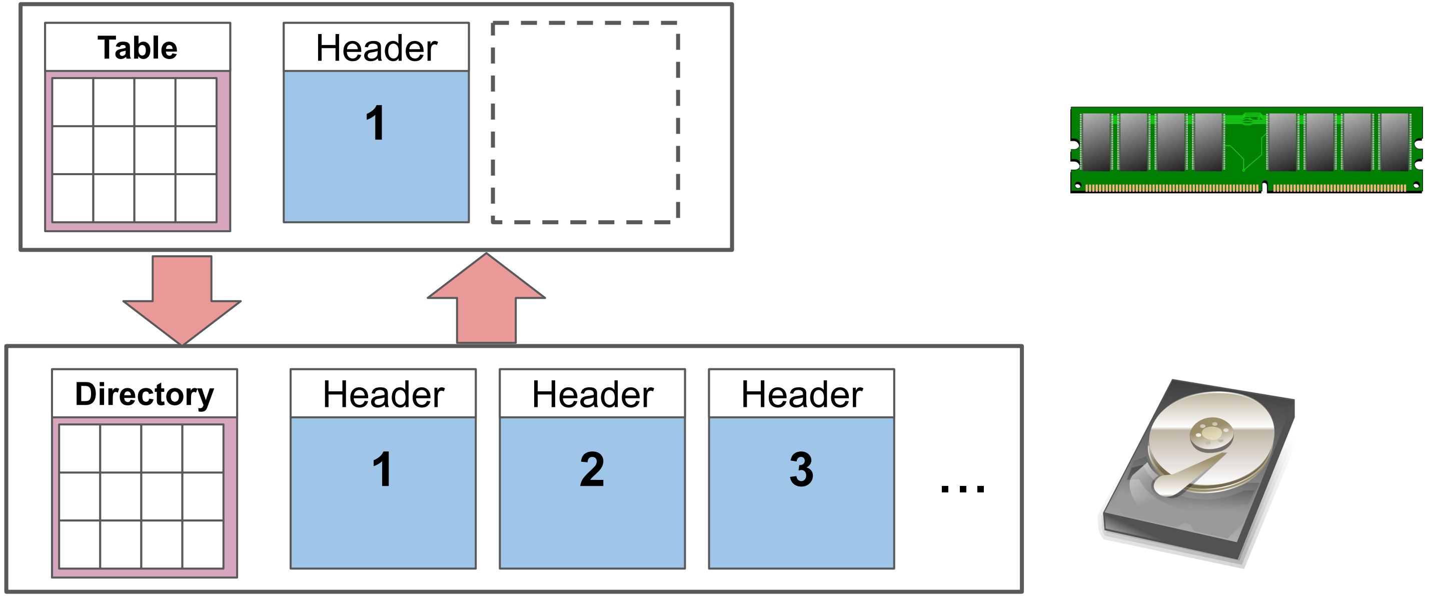
Column Store

+ Fast reads (compression)
+ Fast analytics (SUM, etc.)

monetdb

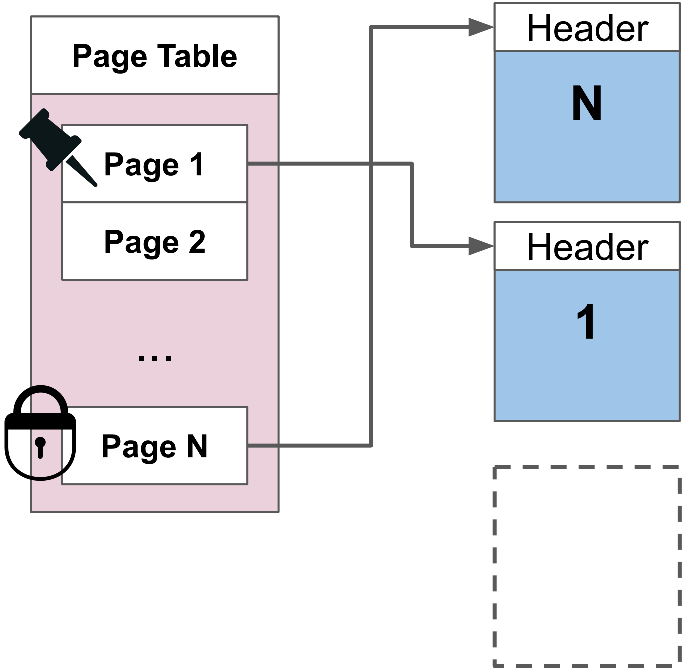
Buffer Pool

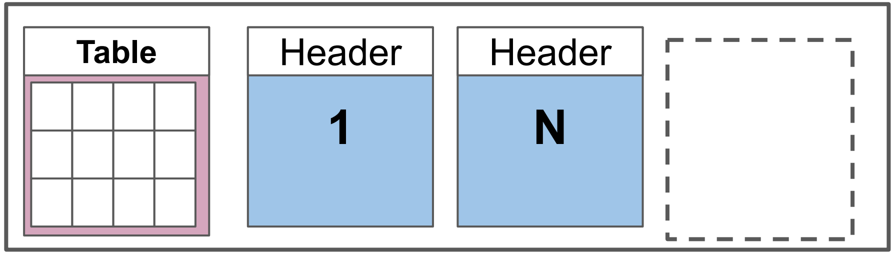
- Problem: how to manage the limited amount of memory
  - How to move data back and forth from disk



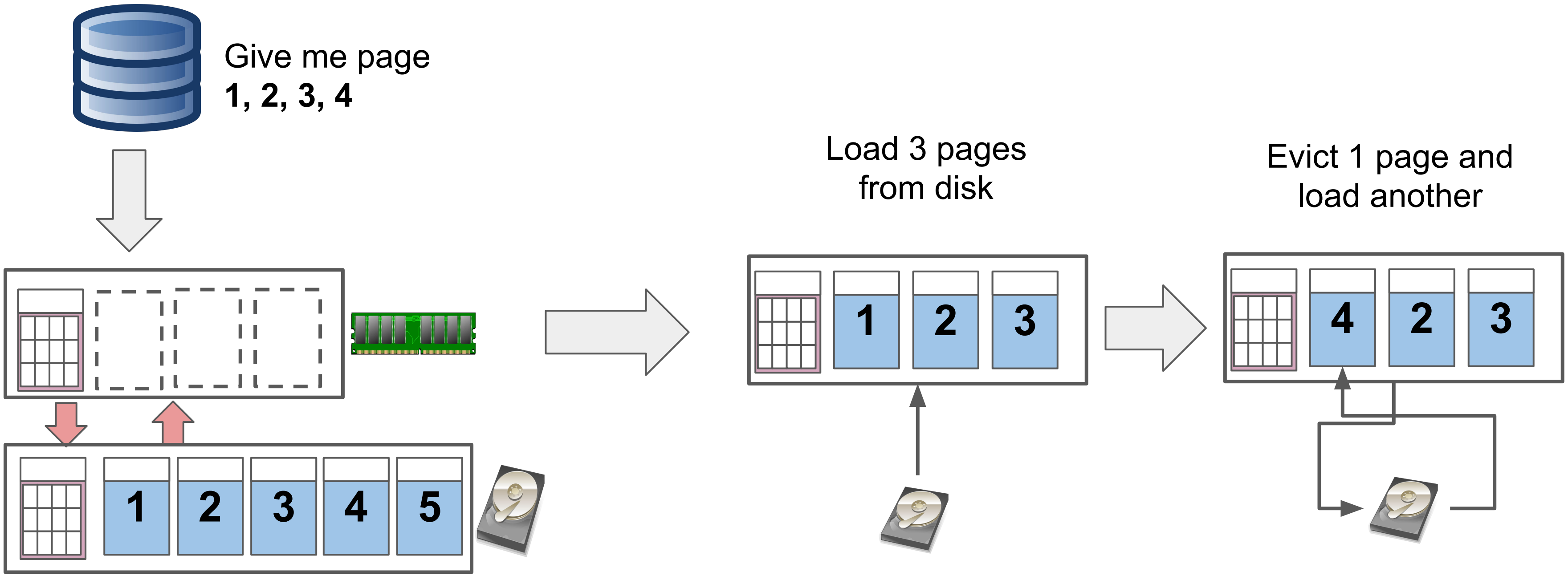
Buffer Pool

- Page Table:
    - Pointer to memory page
    - And other metadata: pin, lock, etc.





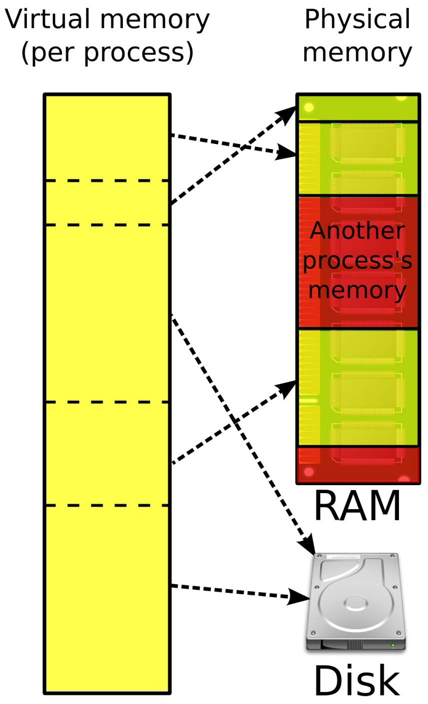
Buffer Pool


Give me page
1, 2, 3, 4
1
2
3
4
5
Load 3 pages
from disk
1
2
3
Evict 1 page and
load another
4
2
3

Buffer Pool

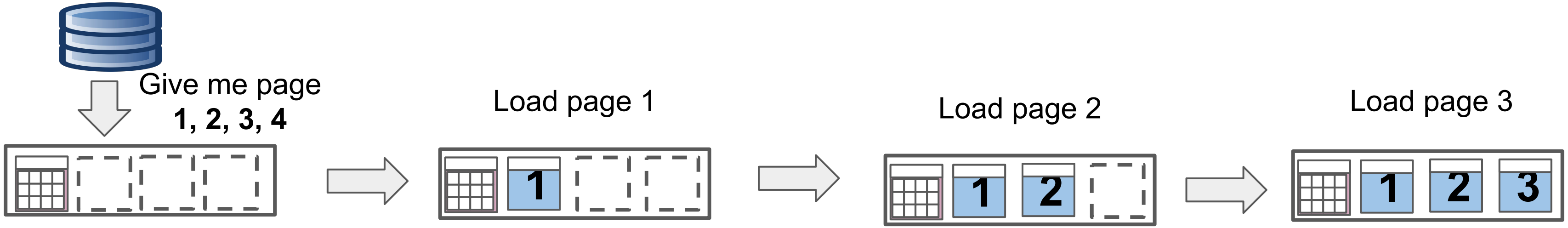
VM provides some mechanism for paging out and paging in but OS does not know anything about the application so they cannot really optimise the replacement policy (as much).

- Virtual Memory already does it!
- True, but:
  - OS knows nothing about the application
  - DBMS knows
- DBMS:
  - Know the access pattern → it can:
    - Pretech pages into memory
    - Better replacement policy



Buffer Pool

- OS is **<u>not</u>** your friend
  - On-demand → doesn't know prefetching
  - Every fetch counts!



Buffer Pool

gave only upper bound, no lower bound

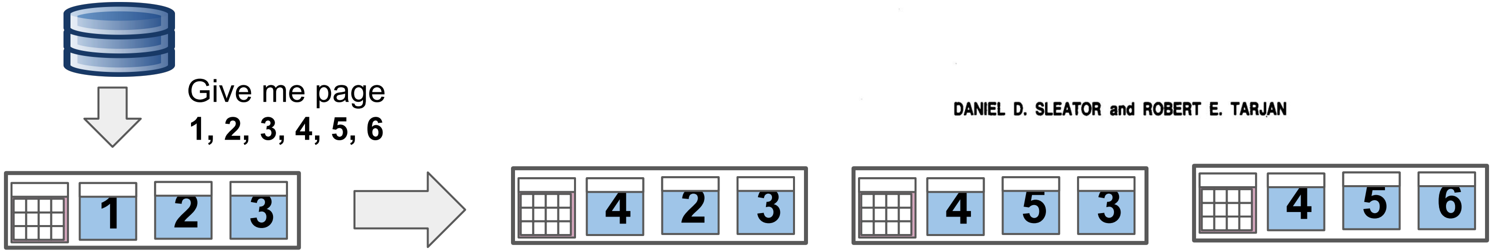
- OS is **<u>not</u>** your friend
  - Buffer replacement policy
  - Optimal policy:
    - Evict one requested farthest in the future
  - OS implements LRU: at most 2x worse than optimal, **without knowing** the future.

Programming Techniques and Data Structures

Ellis Horowitz Editor

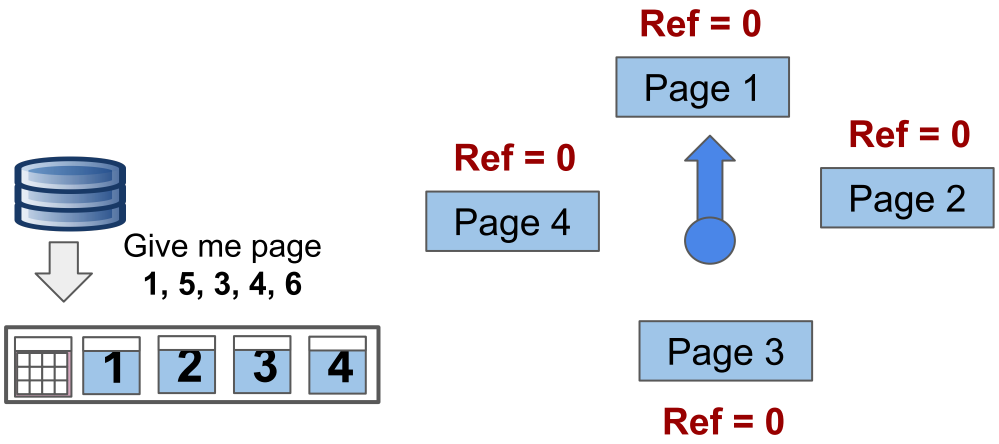
Amortized Efficiency of List Update and Paging Rules

DANIEL D. SLEATOR and ROBERT E. TARJAN



Buffer Pool

- LRU so popular
  - Worth knowing how to implement it



Each page has a reference bit, initially 0

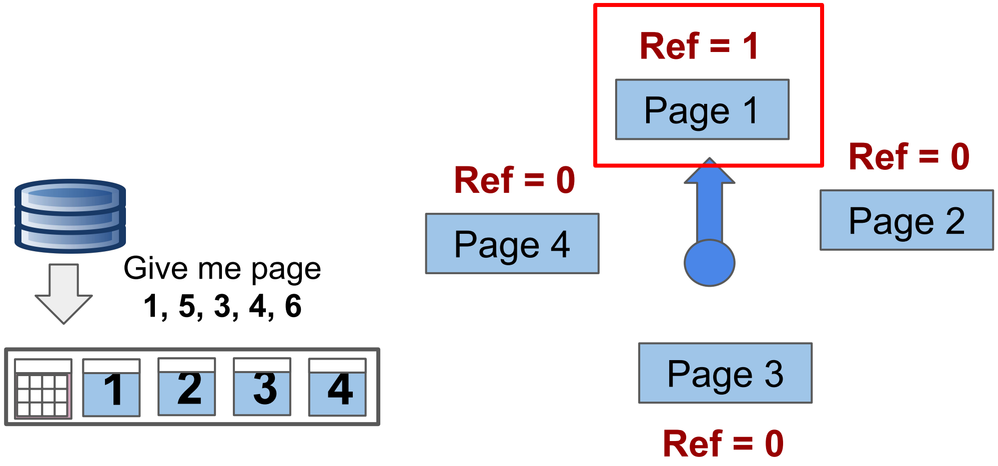
When accessed, set ref = 1

Organized in a circular buffer, with a clock hand (point to a candidate for eviction)

- If a ref bit is 1, set to 0, move on
- Else, evict

Buffer Pool

- LRU so popular
  - Worth knowing how to implement it



Each page has a reference bit, initially 0

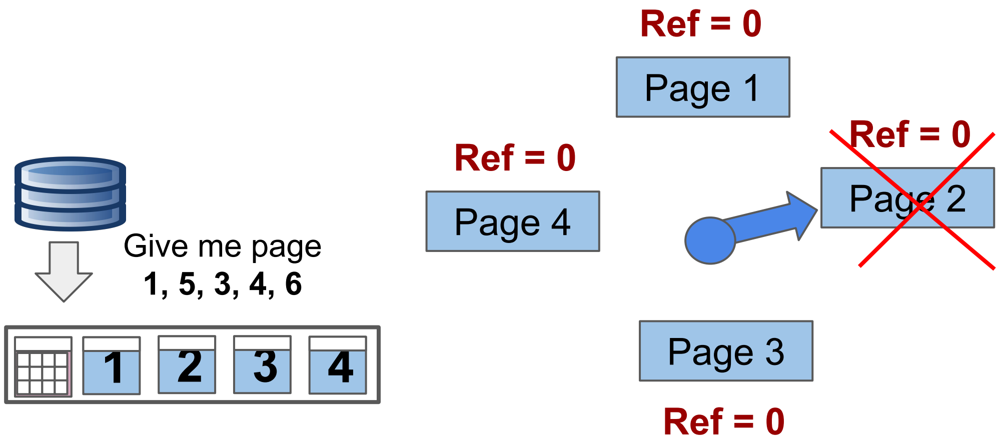
When accessed, set ref = 1

Organized in a circular buffer, with a clock hand (point to a candidate for eviction)

- If a ref bit is 1, set to 0, move on
- Else, evict

Buffer Pool

- LRU so popular
  - Worth knowing how to implement it



Each page has a reference bit, initially 0

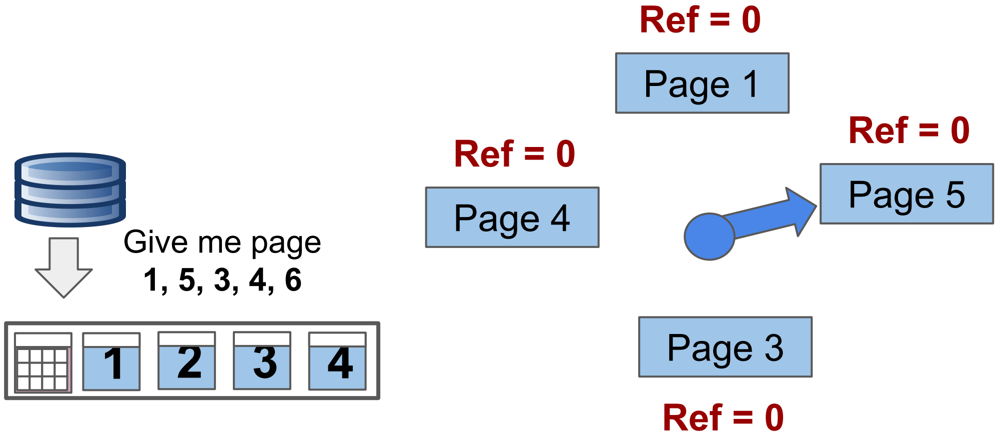
When accessed, set ref = 1

Organized in a circular buffer, with a clock hand (point to a candidate for eviction)

- If a ref bit is 1, set to 0, move on
- Else, evict

Buffer Pool

- LRU so popular
  - Worth knowing how to implement it



Each page has a reference bit, initially 0

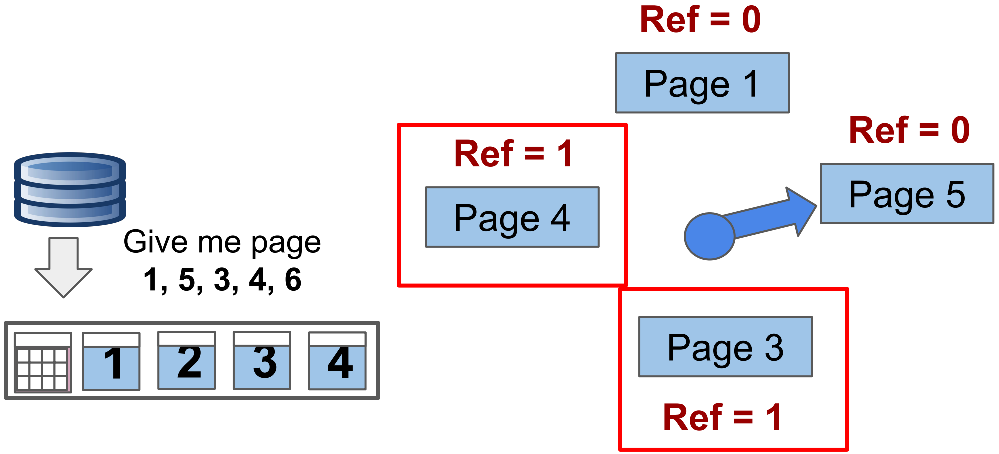
When accessed, set ref = 1

Organized in a circular buffer, with a clock hand (point to a candidate for eviction)

- If a ref bit is 1, set to 0, move on
- Else, evict

Buffer Pool

- LRU so popular
  - Worth knowing how to implement it



Each page has a reference bit, initially 0

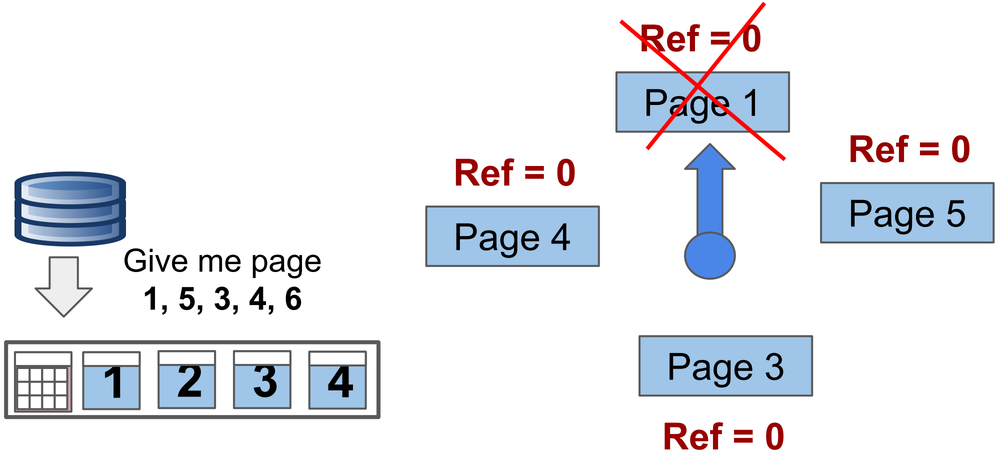
When accessed, set ref = 1

Organized in a circular buffer, with a clock hand (point to a candidate for eviction)

- If a ref bit is 1, set to 0, move on
- Else, evict

Buffer Pool

- LRU so popular
  - Worth knowing how to implement it



Each page has a reference bit, initially 0

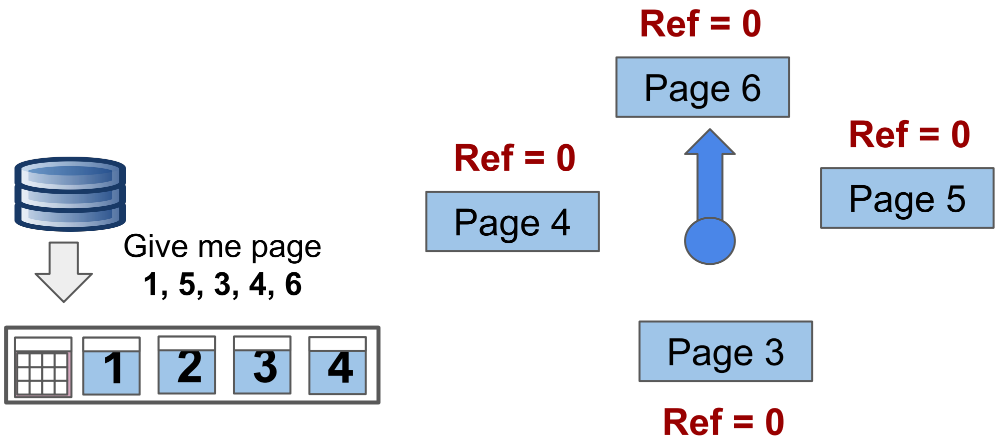
When accessed, set ref = 1

Organized in a circular buffer, with a clock hand (point to a candidate for eviction)

- If a ref bit is 1, set to 0, move on
- Else, evict

Buffer Pool

- LRU so popular
  - Worth knowing how to implement it



Each page has a ref bit, initially 0

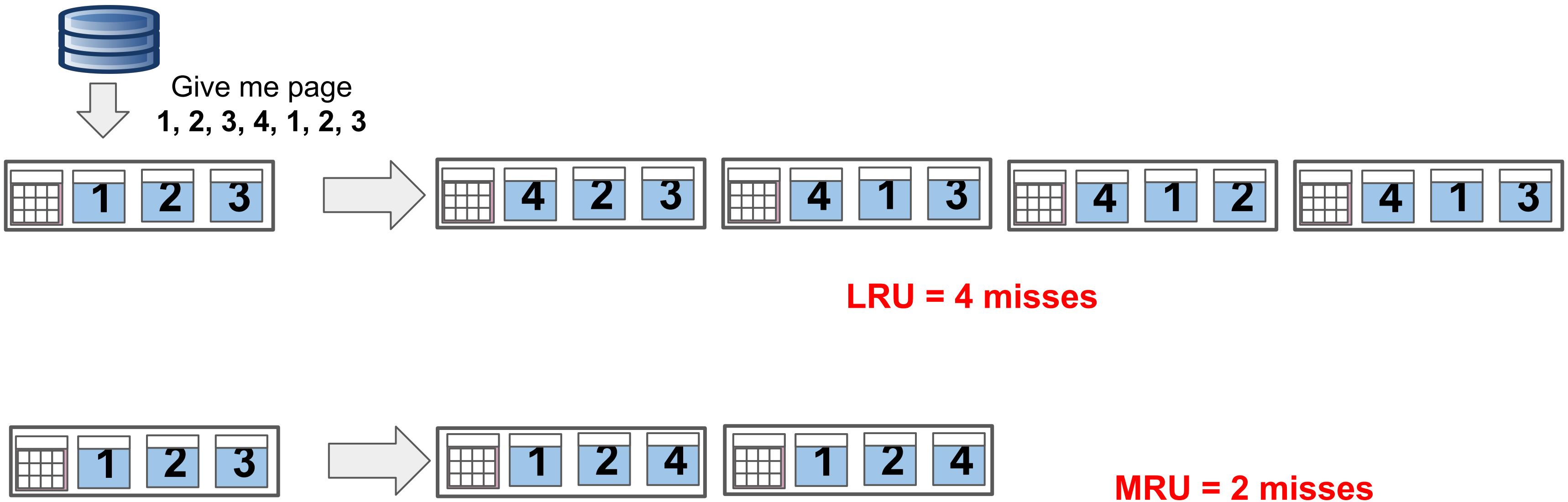
When accessed, set ref = 1

Organized in a circular buffer, with a clock hand (pointing to a candidate for eviction)

- If a ref bit is 1, set to 0, move on
- Else, evict

Buffer Pool

- But LRU isn't always good



Buffer Pool

- You can see now that OS is not good at managing DBMS memory pages
- DBMS exploits access patterns to do better
    - Sequential scan: 1→2→3→4→..
    - Tree (index) scan: 1→4→11→1→4→12→…
    - Random (hash lookup): 13→7→1→4→…

LRU

MRU

Any

Buffer Pool

SQL Query

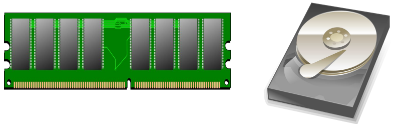
Hints on access pattern

Buffer Pool Manager

Allocation Policy

Eviction Policy

Prefetch Policy



- Bypass OS's Page Cache: O_DIRECT
- Most DBMS use multiple buffer pools:
  - Per database
  - Per page type

Summary

- Database stores data in files
- Disk Manager decides page layout on disk
- Buffer Manager moves pages in and out of memory
- OS is **<u>not</u>** your friend