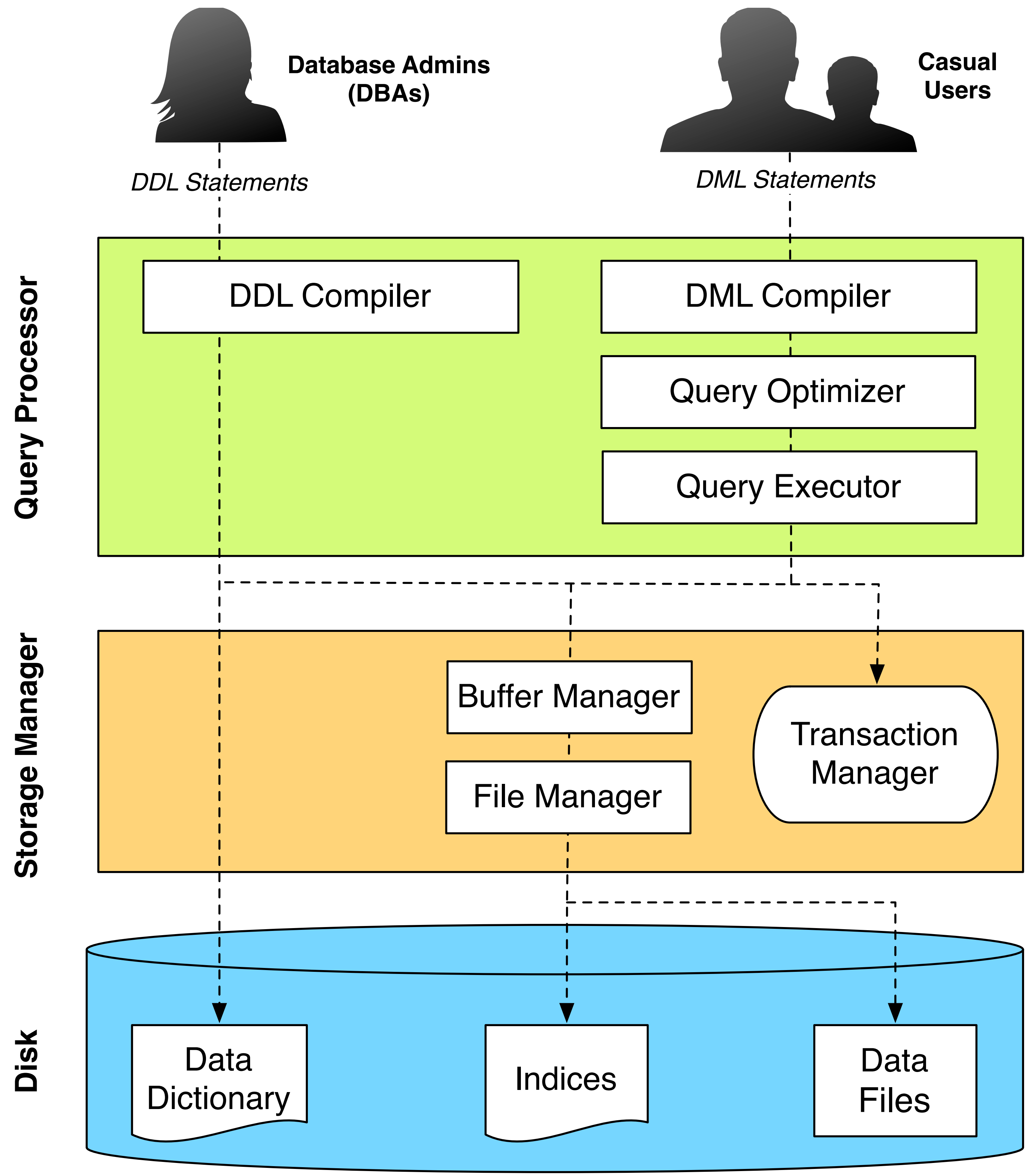
CS 455
Principles of Database Systems

UNIVERSITY of PUGET SOUND
Est. 1888

Department of Mathematics
and Computer Science

Lecture 5
Persistent Storage and Files

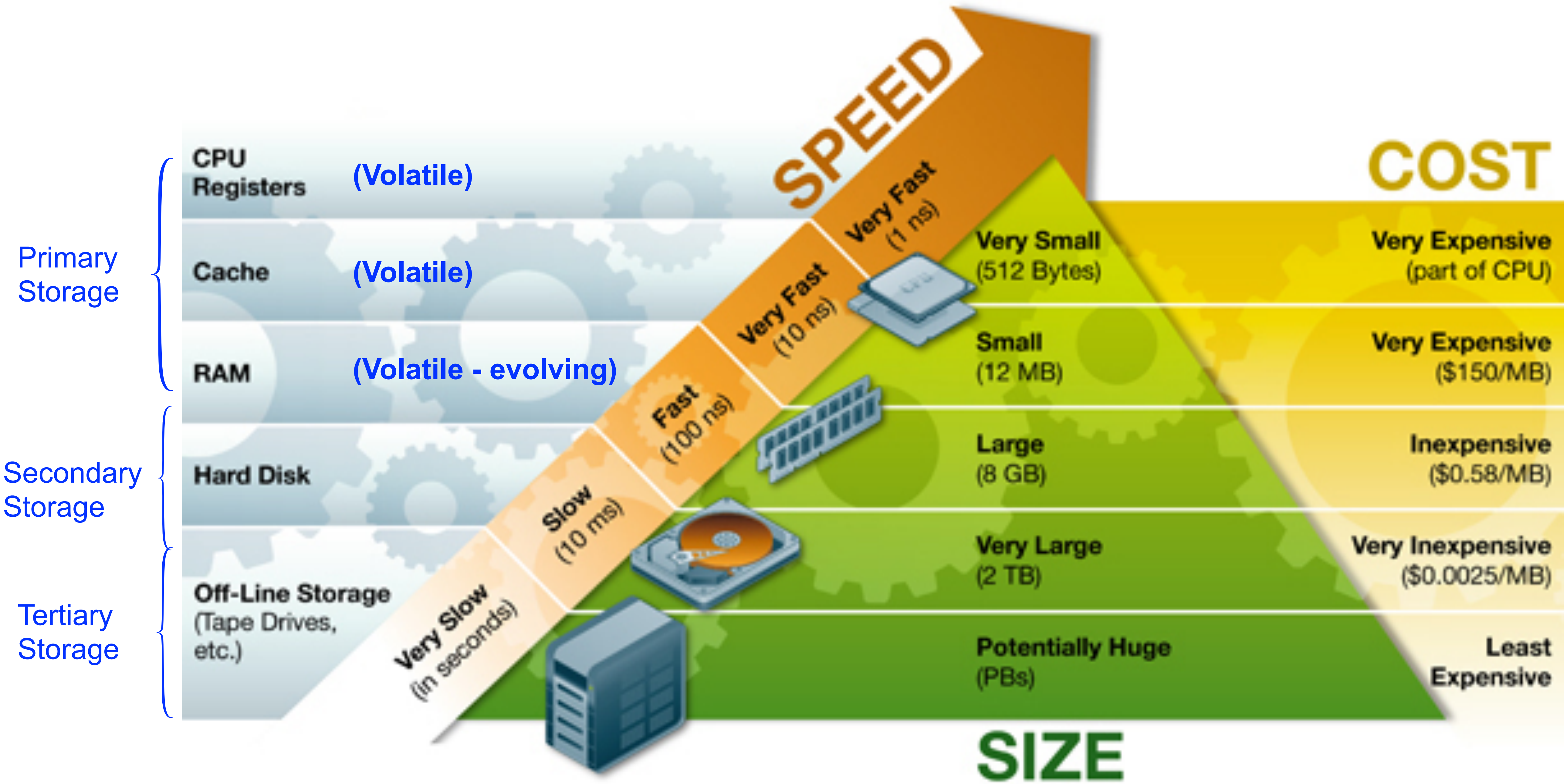
Relational Database System Architecture



Topics

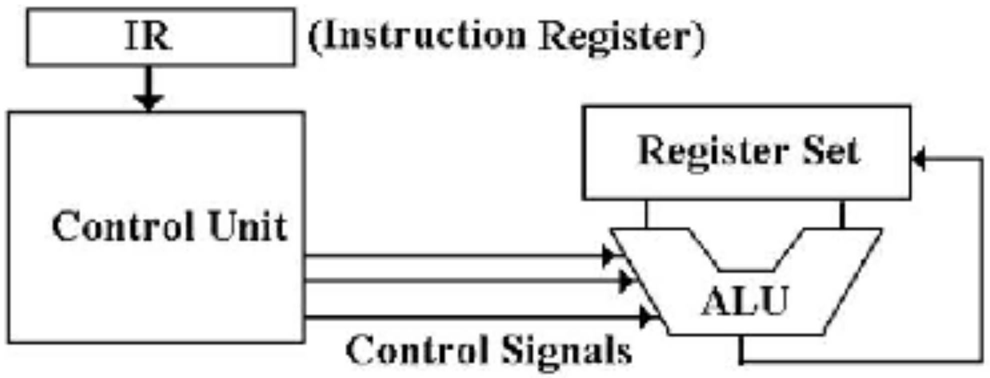
- Disks
  - Hardware
  - Understanding Files and Blocks
  - Modeling Access Time
  - Disk Scheduling
- Buffer Management

Storage (Memory) Hierarchy



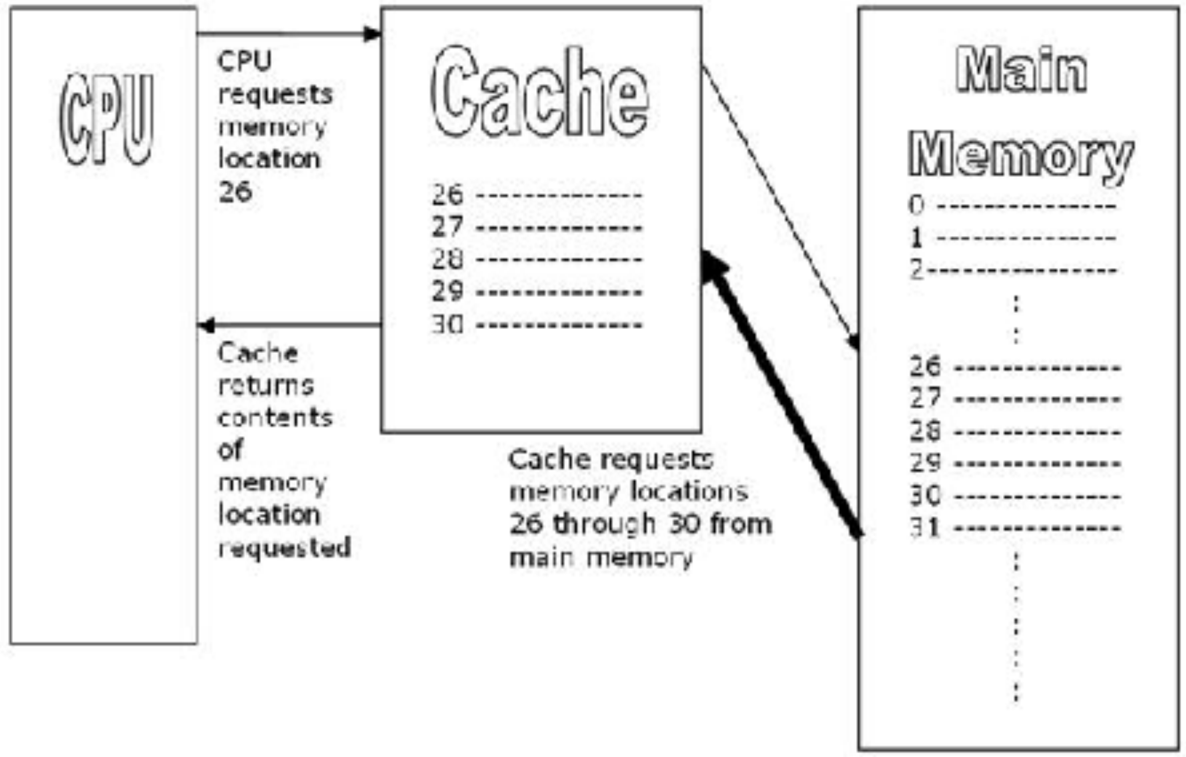
Storage Hierarchy: CPU Registers

- *CPU Registers (Volatile!)*
  - Fastest level of storage and the closest to computation
    - The CPU's arithmetic logic unit (ALU) performs calculations and comparisons directly off register data
- Each register stores 32 or 64 bits (of instruction or data)
  - Only a few registers, even on modern CPUs
    - ~32 integer registers, 32 floating point registers for MIPS CPUs
    - ~8 on x86 (Intel CPUs)
    - Why not pack more?

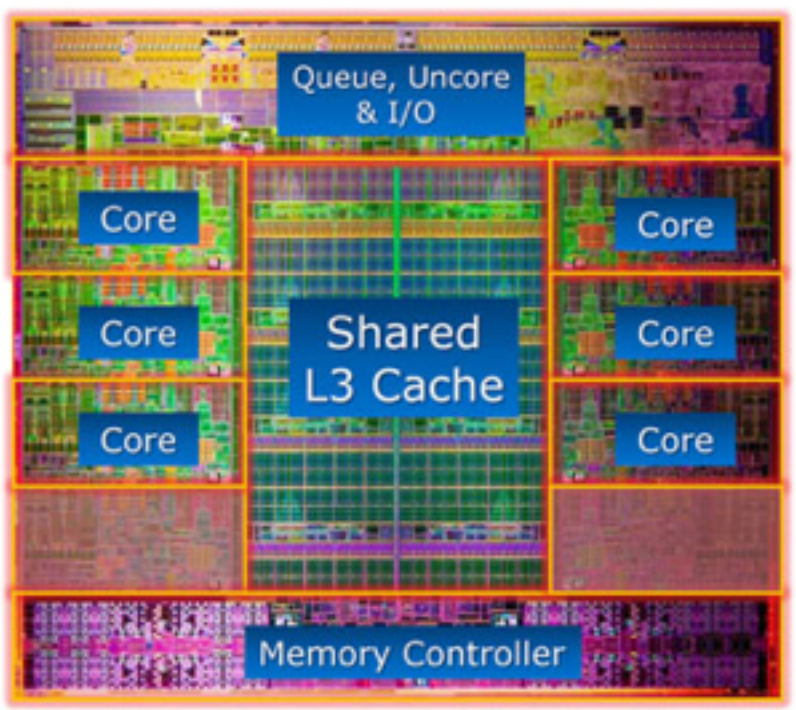


Storage Hierarchy: Cache

- *CPU Cache (Volatile!)*
  - A few KBs to several MBs
  - Resides on CPU
    - Modern CPUs have several “levels” of cache before exploring memory
    - Levels: L1, L2, and L3 are common
    - Sizes: L1 < L2 < L3
  - Automatically managed by hardware
    - We can’t program caches to do what we want... but we can be better programmers



Intel® Core™ i7-3960X Processor Die Detail



Storage Hierarchy: Main Memory

UNIVERSITY of PUGET SOUND

- *[Main Memory](#) (Volatile!)*
  - Several GBs
    - New iMacs: 32 GB RAM
  - Still fast(-ish): 10-100 nanoseconds (ns)
  - Still quite costly
    - $0.0054 per MB

      http://www.jcmit.com/memoryprice.htm

- In general, still too small and expensive to hold DBs
  - Also, no "data persistence" at this level

Storage Hierarchy: Hard Drives

- *Hard Drives (Non-Volatile!)*
  - A few TBs
    - New 2013 iMacs: 1 TB
    - Our HPC cluster node: 4 TBs (Two 2 TB disks)
  - Very cheap
    - $0.0000467 per MB

      http://www.jcmit.com/diskprice.htm

- A DB is stored here, but...
  - Hard drives are slow: Few milliseconds (ms)
    - Why so slow?
    - We need to understand how it's built

Storage Hierarchy: Solid-State Disks

- *Solid State Disks or SSDs (Nonvolatile!)*
  - Hundreds of GBs to a few TBs
  - Faster than magnetic disks: 100 microseconds (us)
  - An order of magnitude more expensive than magnetic drives
    - $0.000518 per MB

      http://www.jcmit.com/flashprice.htm

- Is *write endurance* a problem?
  - A block of memory can endure 2000 to 3000 write operations

"This breed of non-volatile storage retains data by trapping electrons inside of nanoscale memory cells. A process called tunneling is used to move electrons in and out of the cells, but the back-and-forth traffic erodes the physical structure of the cell, leading to breaches that can render it useless."

http://techreport.com/review/27909/the-ssd-endurance-experiment-theyre-all-dead

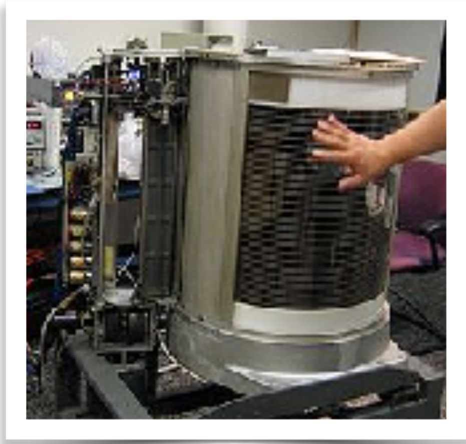
Storage Hierarchy: Offline Storage

- *Offline (Tertiary) Storage (NonVolatile)*
  - Optical drives, tapes, removable flash, cloud storage services (S3, DropBox)
  - Enormous amounts of offline storage
  - Extremely cheap and expendable

- Way too slow for production DB applications
  - But are still relevant today
  - Mostly for backup and portability

Today: Magnetic Disks Still Dominate

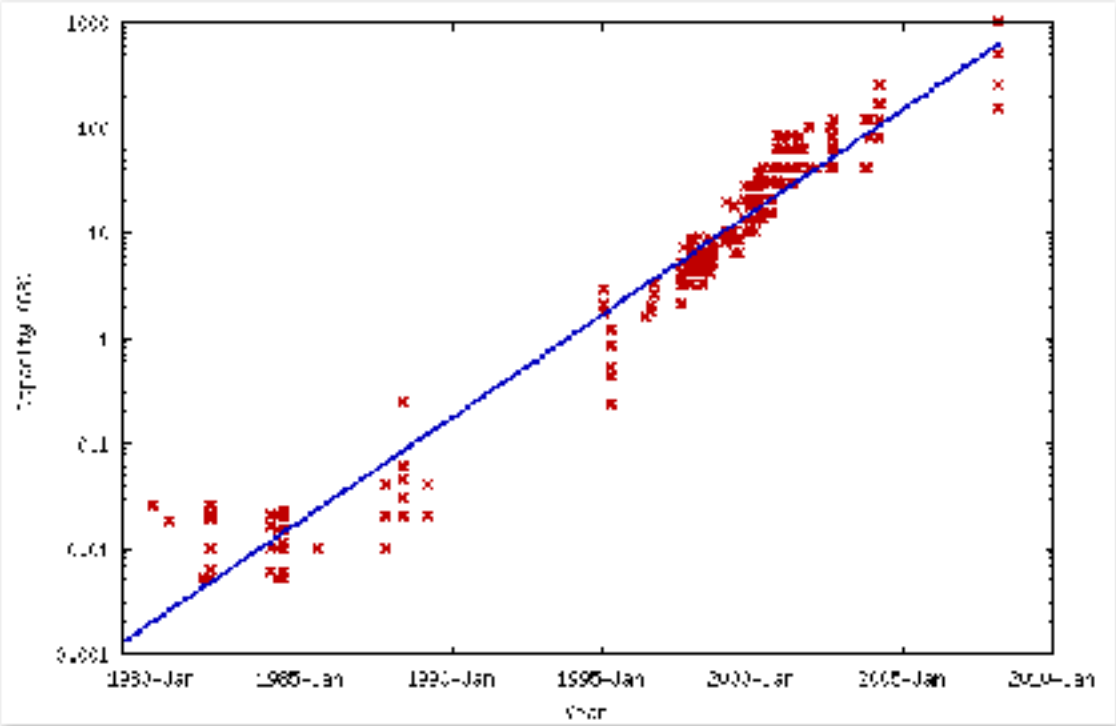
- First modern magnetic HDD with moving head
  - Shipped with IBM 305 RAMAC (1956)
  - Held 5 MB for a whopping $10,000 per MB!
  - 24 inches diameter, single-head

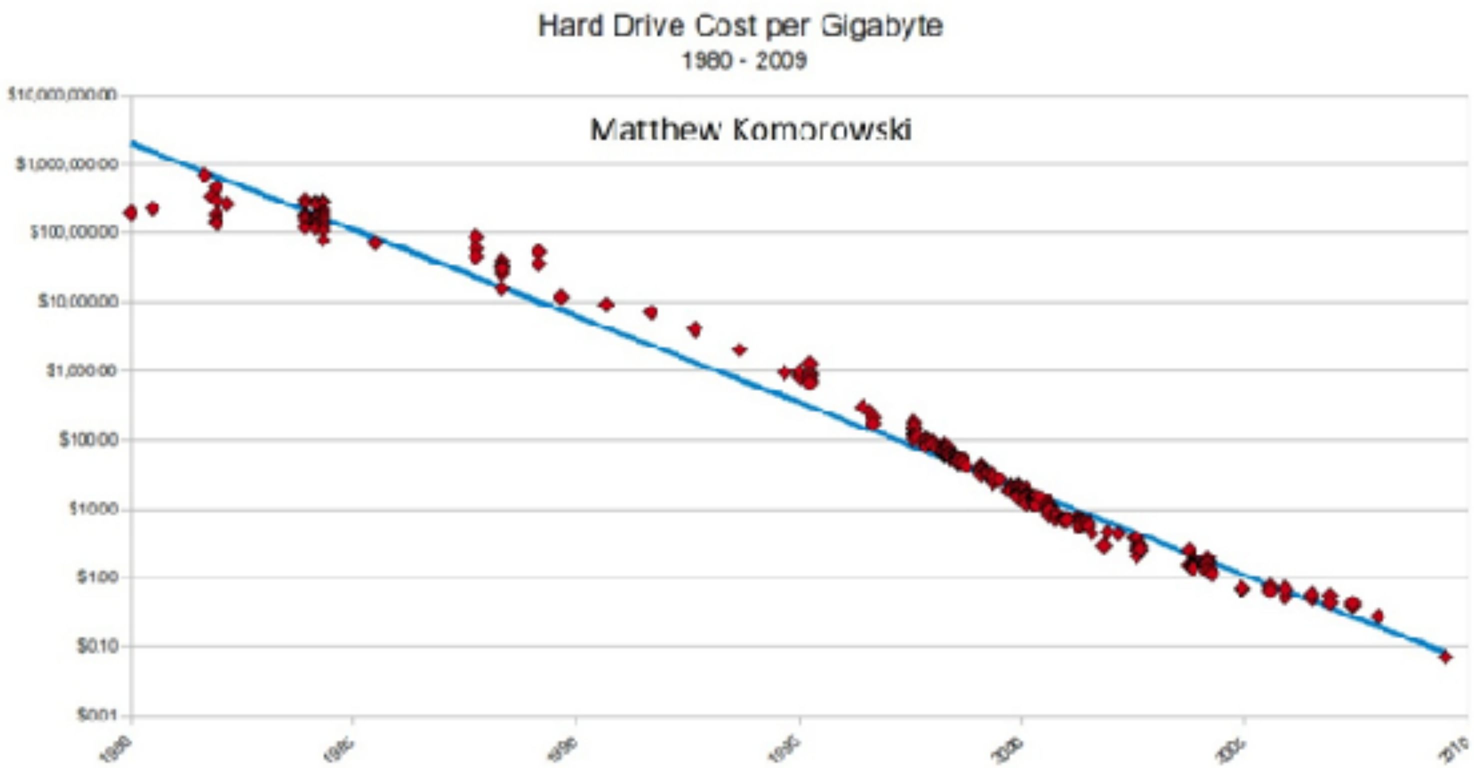




- Newest HDD today (2021)
  - 10 TB, 7200RPM, 3.5 inches under $300 total < $0.0000003 per MB
- Full History
  - http://www.pcworld.com/article/127105/article.html

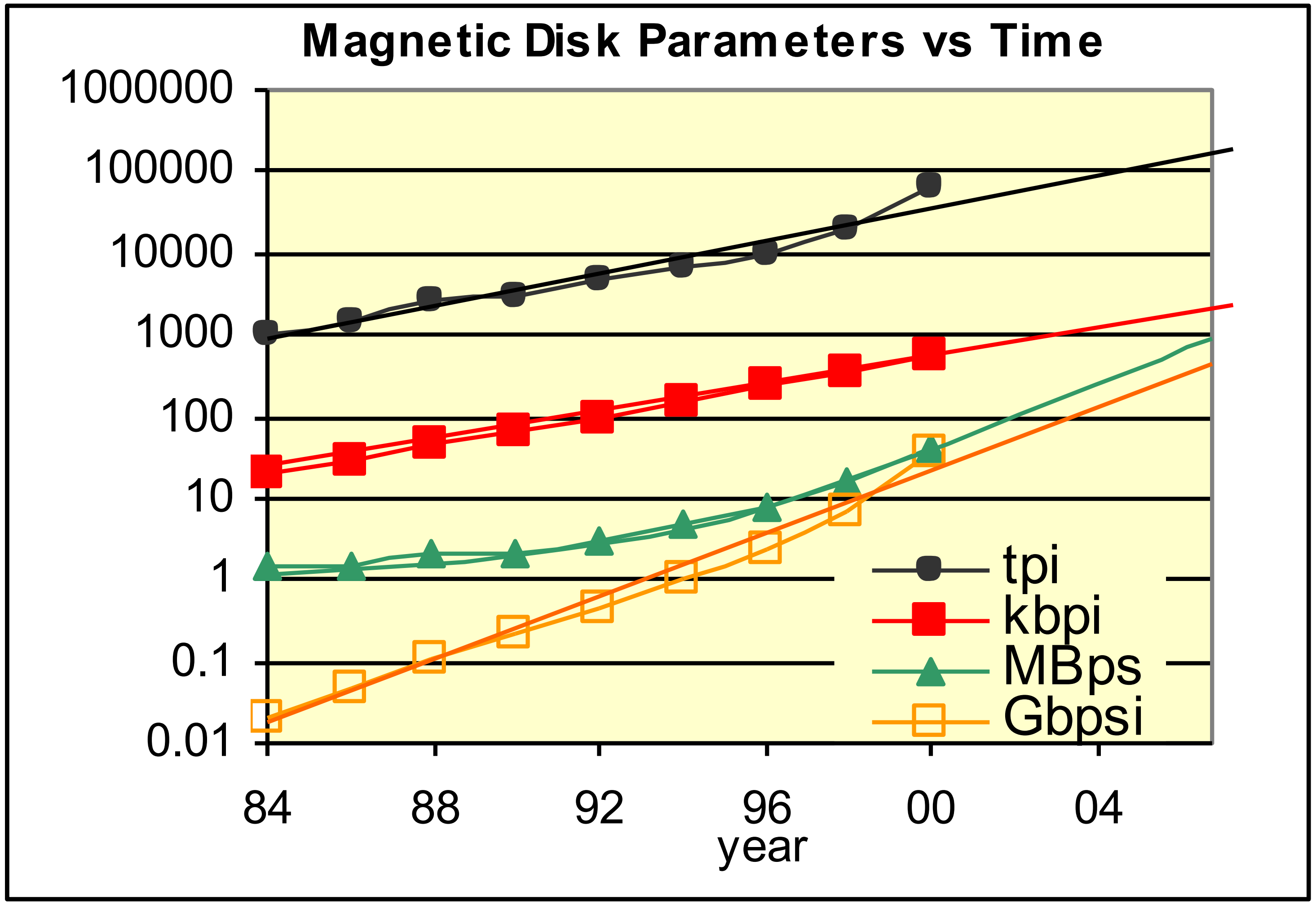
Magnetic Disk Growth

- Disks over the years:
  - Exponentially dense
  - Exponentially cheaper per MB stored
  - But what about disk performance?





Magnetic Disk Growth (Cont.)

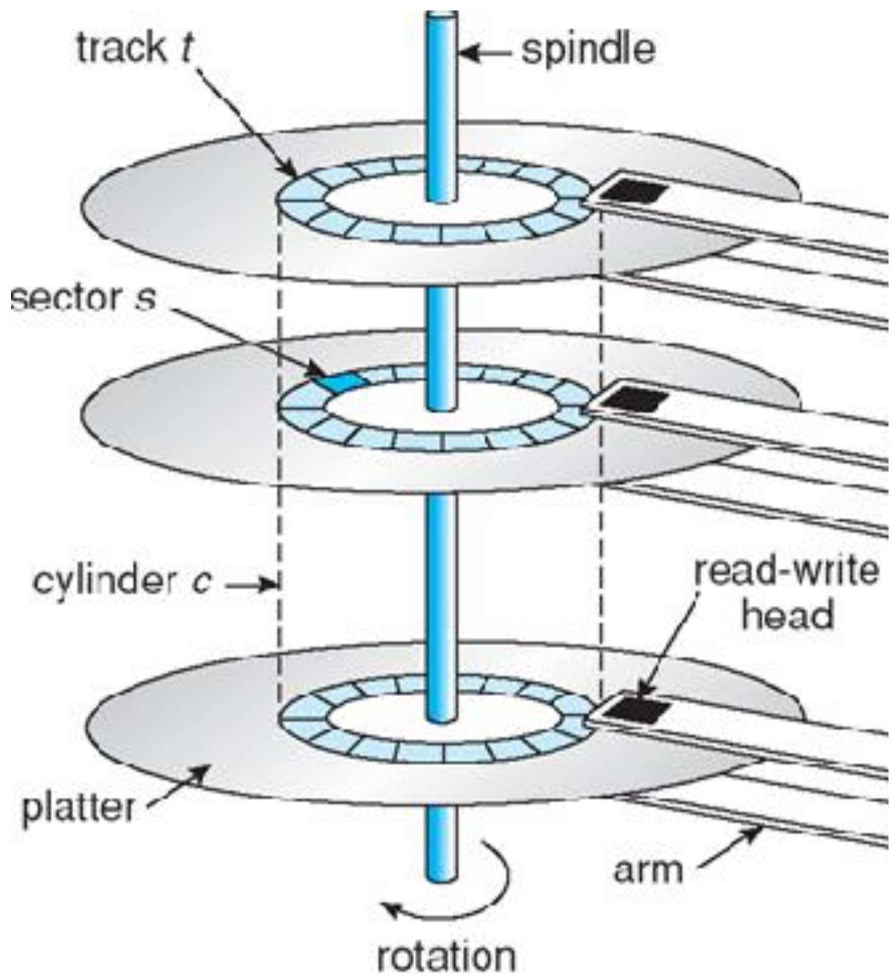


Credits: Jim Gray. "Rules of Thumb in Data Engineering."

- Disk capacity grows 60% per year
- Disk performance grows 10 times slower!
  - 7% per tear
- Legend:
  - tpi = tracks per sq-inch
  - kbpi = kilobits per sq-inch
  - MBps = MB/sec

Hard Disk Drives (HDD)

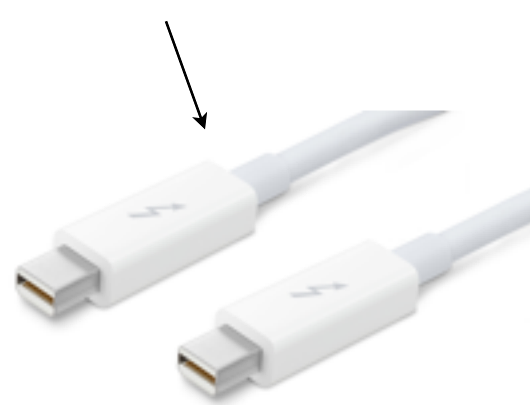
- A disk has of one or more *platters*
  - Each *platter* has two *surfaces*
  - Each *surface* has many *tracks*
    - 2 to 300,000 tracks per surface
  - Each *track* has many *sectors*
  - Each *sectors* stores **512** Bytes
- Each *platter* has a *head* that performs read/write operations
- *Cylinders* are the set of the same tracks on all surfaces



Communication Media

- Disks are connected to the host machine via a communication medium

- Cables are rated by *transfer rates (or bandwidth)*:
  - IDE (ATA): ~2.8 Gbps
  - SATA 3 (Serial ATA): 6.0 Gbps
  - SCSI (pronounced "scuzzy"): 5.0 Gbps
  - USB 3.0: Originally 5.0 Gbps
    - (Now 10 Gbps to compete with Apple Thunderbolt)

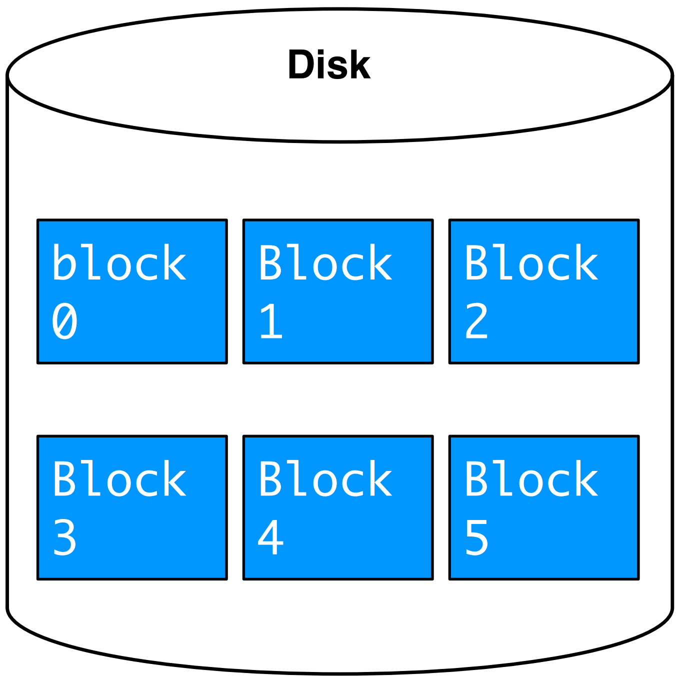


Topics

- Disks
  - Hardware
  - Understanding Files and Blocks
  - Modeling Access Time
  - Disk Scheduling
- Buffer Management

Disk Blocks

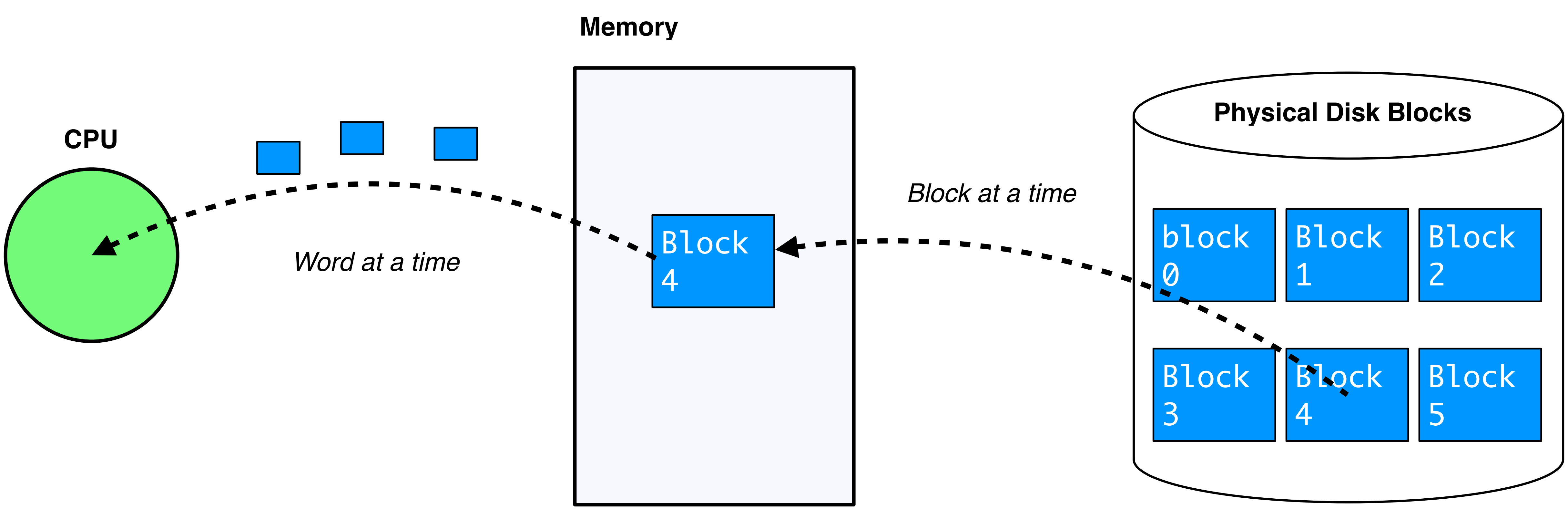
- *Block:* Contiguous sequence of sectors from a single track
  - Data is read/written a block at a time
  - Standard block size is usually 4 KB
    - (8 * 512 Byte sectors)



- **Important!** Assume *blocks* are the basic unit of data transfer
  - Not sector
  - Even though "sector size" often synonymous with "block size"

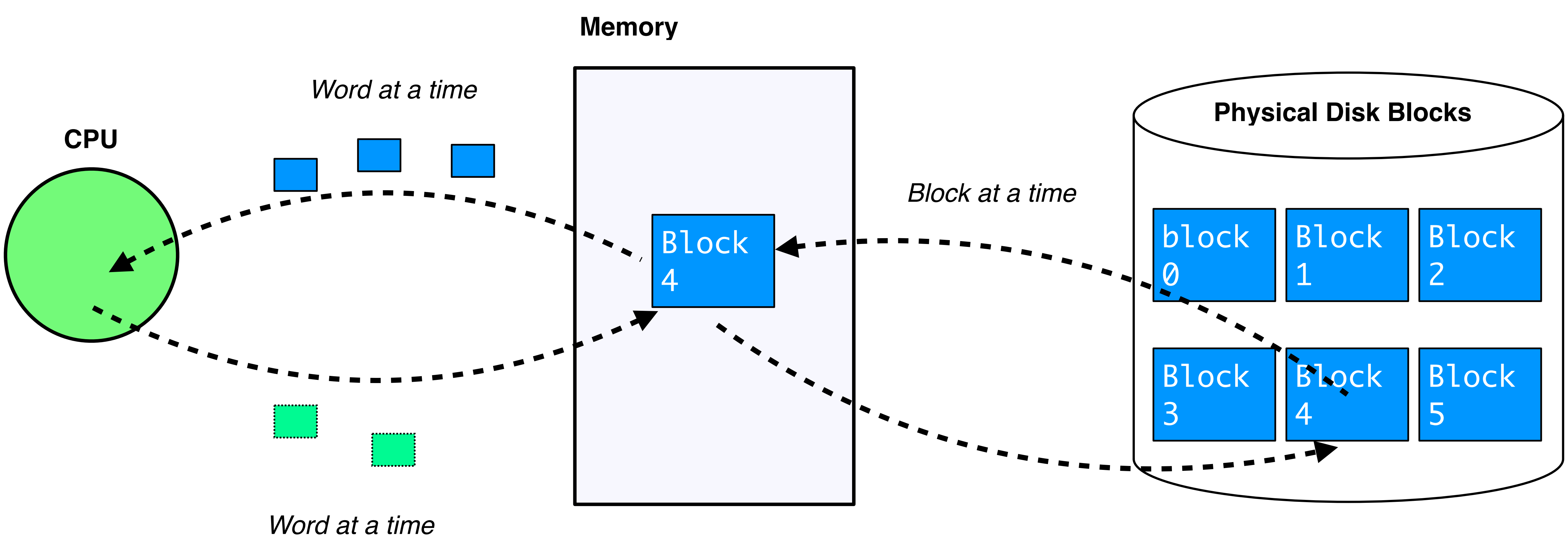
Translating User to System View

- What happens if user says, "**read** bytes 2 - 14 from a file?"
  - OS will fetch disk block(s) corresponding to those bytes
  - Return just the correct portion of the block(s) to CPU
  - Assume bytes 2-14 in the file is in **Block 4** on disk



Translating User to System View

- What happens if user says, "**write** bytes 2 - 14 into a file?"
  - OS will fetch disk block(s) corresponding to those bytes
  - Modify portion in memory
  - Write out block to disk:
    - Accesses increases by factor of 2! Writes are expensive!



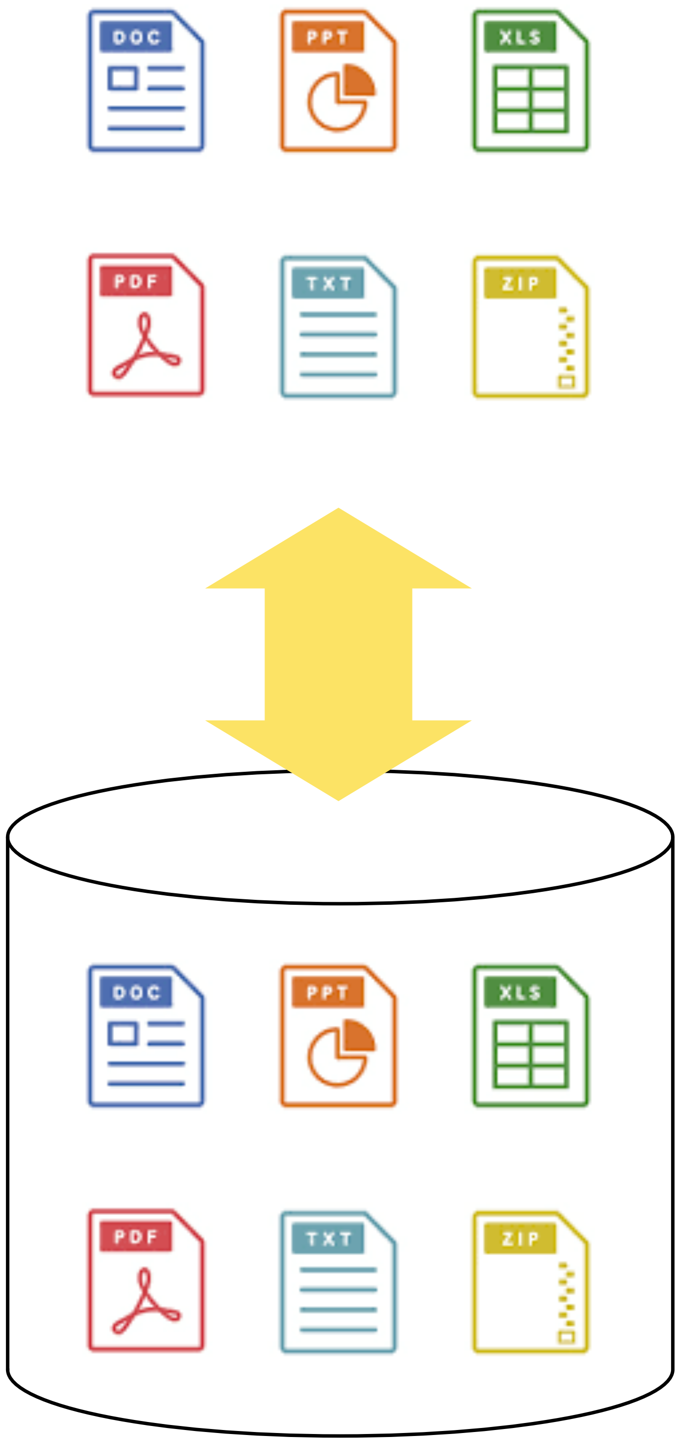
Then What's a File?

- A *File* is just an operating system abstraction for a sequential *stream* of bytes.

Then What's a File?

- A *File* is just an operating system abstraction for a sequential ***stream*** of bytes.

- To us: We ***think*** that a file is just one contiguous unit kept together on disk

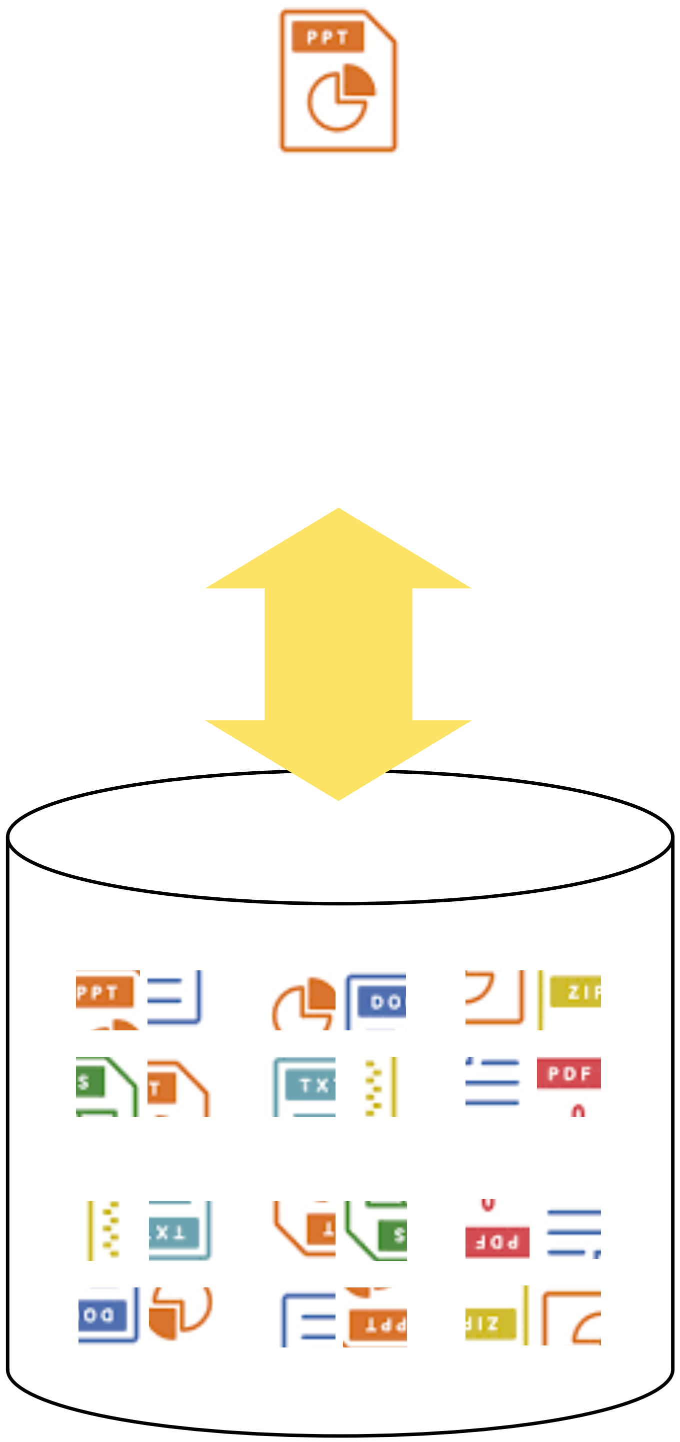


Then What's a File?

- A *File* is just an operating system abstraction for a sequential ***stream*** of bytes.

- To us: We ***think*** that a file is just one contiguous unit kept together on disk

- In reality: Every file is split into blocks
  - Blocks themselves may or may ***NOT*** be arranged contiguously on disk
  - If we're **unlucky**, file blocks are scattered across disk (file is "***fragmented***")

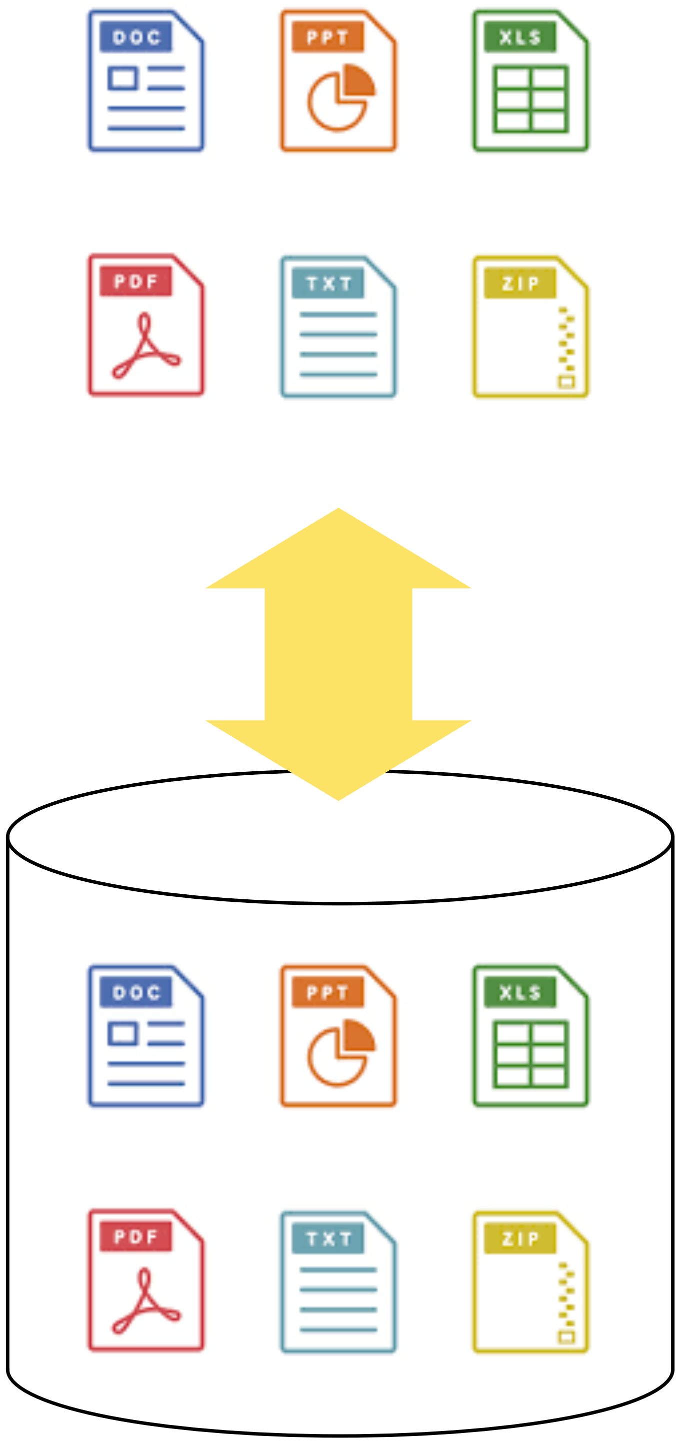


Then What's a File?

- A *File* is just an operating system abstraction for a sequential *stream* of bytes.

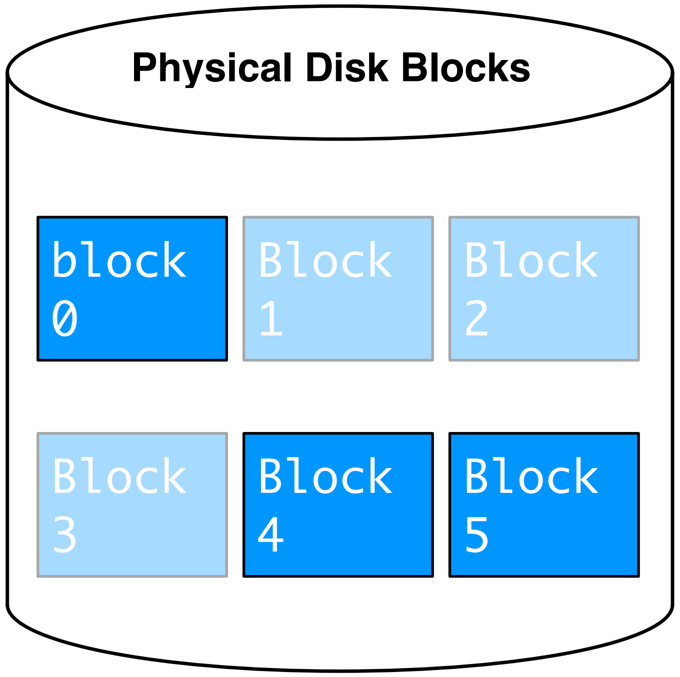
- To us: We *think* that a file is just one contiguous unit kept together on disk

- In reality: Every file is split into blocks
  - Blocks themselves may or may not be arranged contiguously on disk — ***NOT***
  - If we're ***lucky***, the file blocks are also contiguous on disk!



The File Illusion

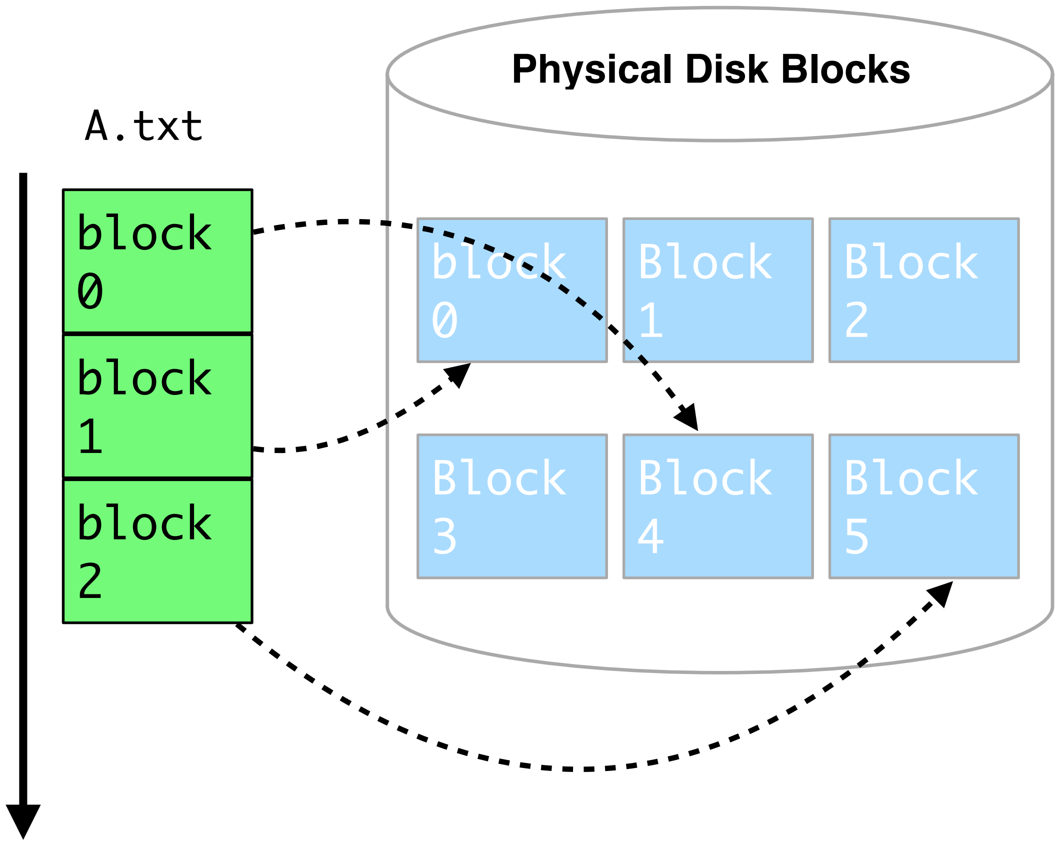
- Example: block size = 4 KB, and some file `A.txt` is 11 KB
  - A.txt requires *ceiling(11 KB / 4 KB) = 3* blocks
    - OS allocates these disk blocks: 4, 0, 5



The File Illusion

- Example: block size = 4 KB, and some file `A.txt` is 11 KB
  - A.txt requires *ceiling(11 KB / 4 KB) = 3* blocks
    - OS allocates these disk blocks: 4, 0, 5
  - User thinks the 3 blocks are contiguous
    - These are called logical blocks: 0,1,2

```java
Scanner file = new Scanner(new File("A.txt"));
while (file.hasNext()) {
   // we think the file data is contiguous!
   // give me the next line/row of the file!
   String s = file.nextLine();
}
```

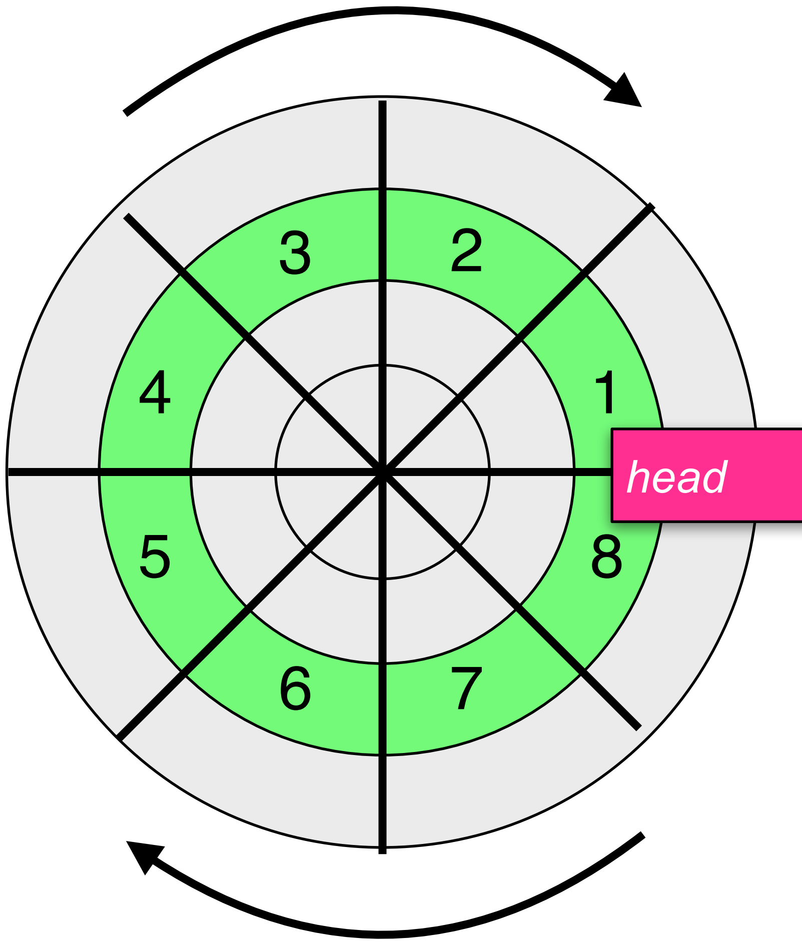


Files are *usually* read sequentially (linear fashion)

Data Access Patterns Matter

- Assume a file's blocks can be arranged contiguously on disk
  - The way we access files (known as *access patterns*) still matter

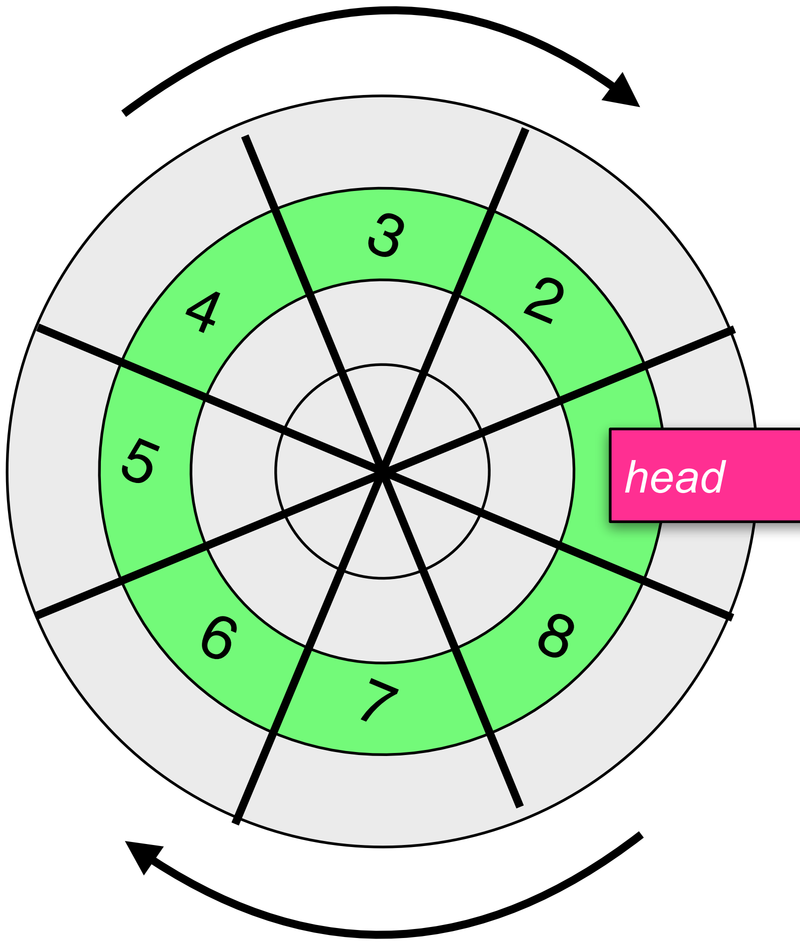
- Access Pattern:
  - How is a file read from (or written to) disk?
    - Sequential access pattern: [1,2,3,4,5,6,7,8]



Data Access Patterns Matter

- Assume a file's blocks can be arranged contiguously on disk
  - The way we access files (known as *access patterns*) still matter

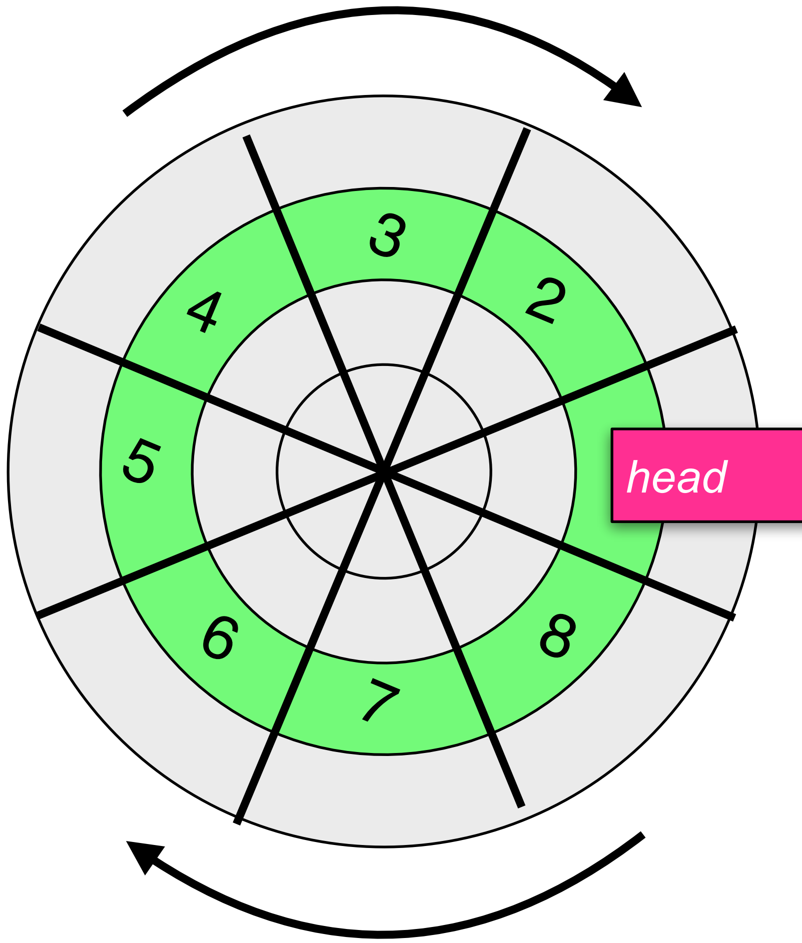
- Access Pattern:
  - How is a file read from (or written to) disk?
    - Sequential access pattern: [1,2,3,4,5,6,7,8]



Data Access Patterns Matter

- Assume a file's blocks can be arranged contiguously on disk
  - The way we access files (known as *access patterns*) still matter

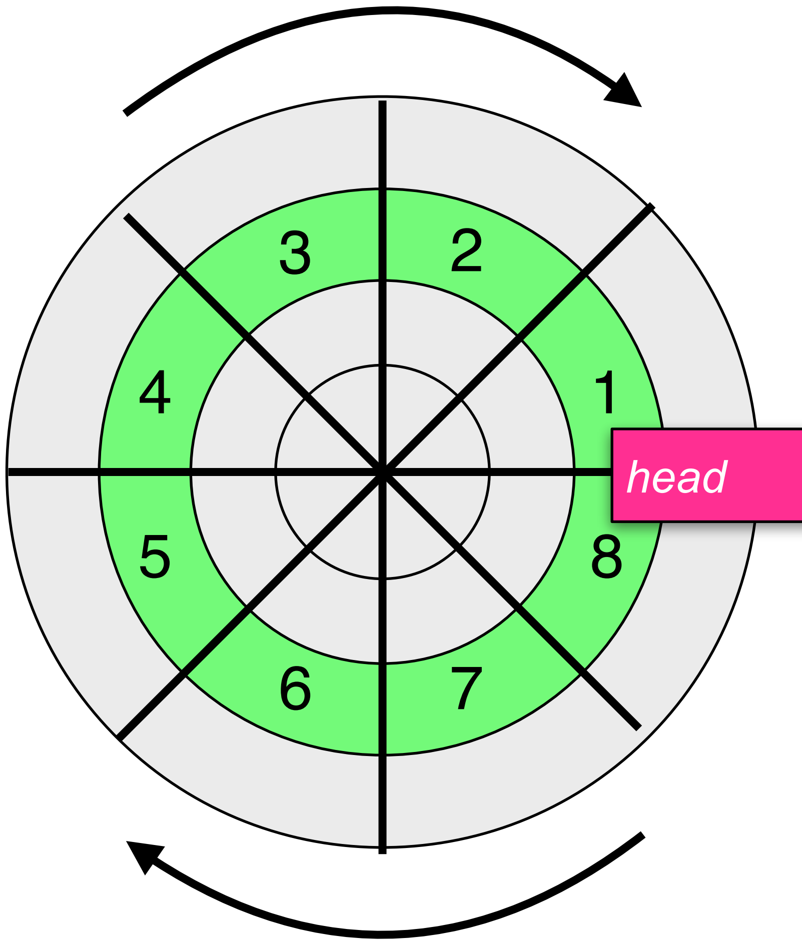
- Access Pattern:
  - How is a file read from (or written to) disk?
    - Sequential access pattern: [1,2,3,4,5,6,7,8]
      - e.g., scanning through the file line-by-line
      - e.g., linear search over a huge array stored in file
      - Ideal situation for performance!



Data Access Patterns Matter

- Assume a file's blocks can be arranged contiguously on disk
  - The way we access files (known as *access patterns*) still matter

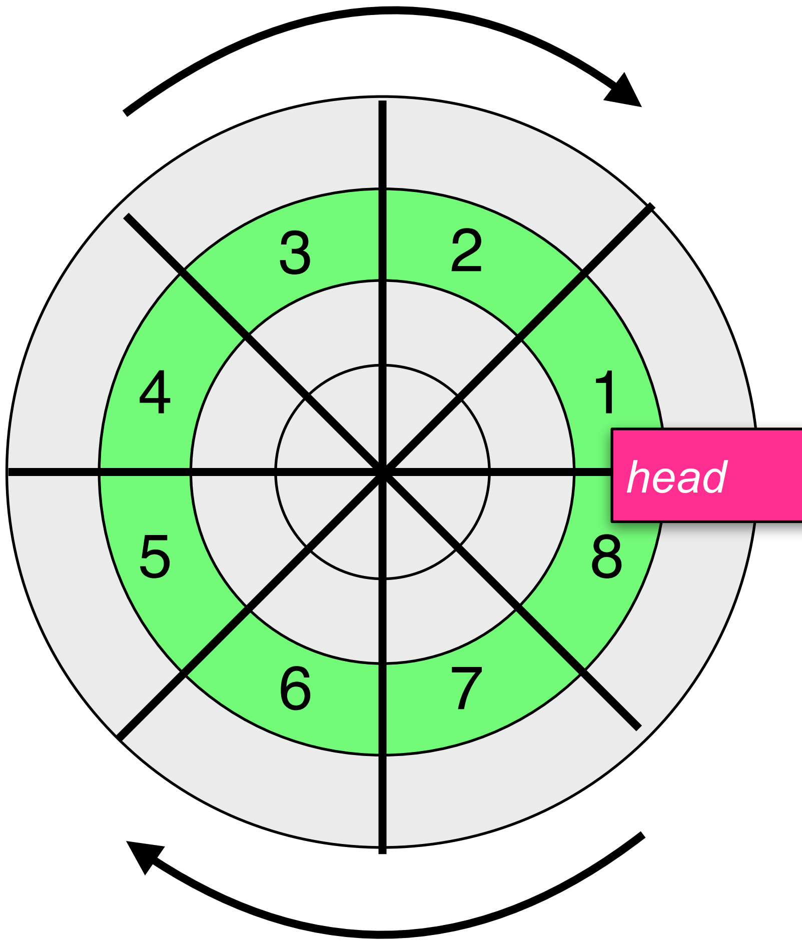
- Access Pattern:
  - How is a file read from (or written to) disk?
    - Random (irregular) access pattern [6,1,8,3,2,4,7,5]



Data Access Patterns Matter

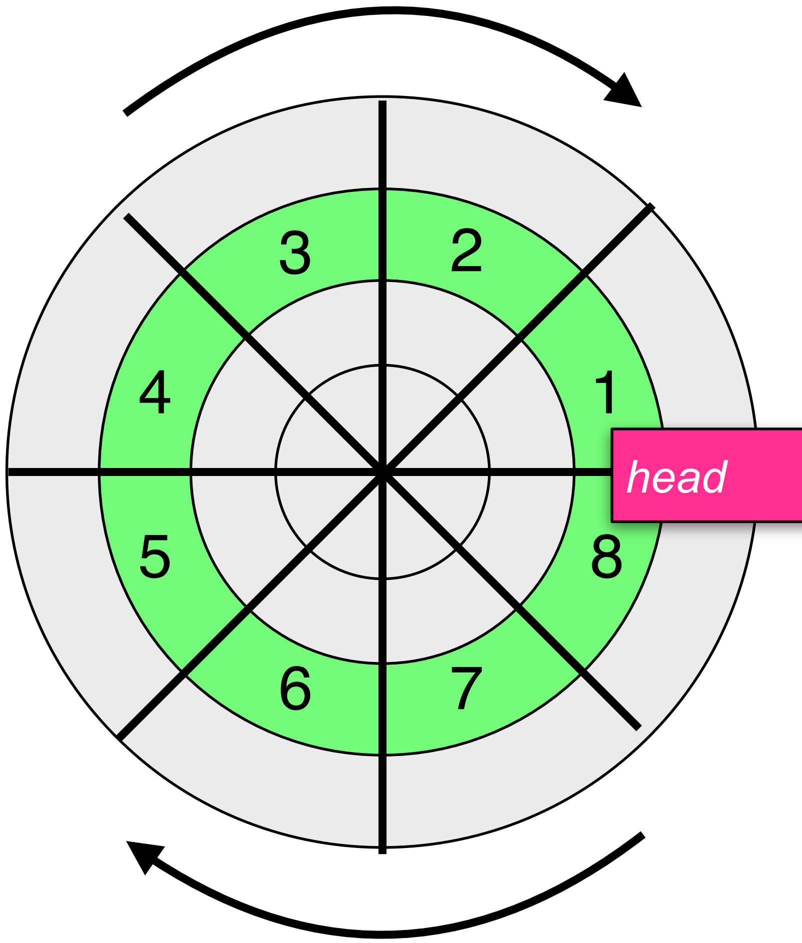
- Assume a file's blocks can be arranged contiguously on disk
  - The way we access files (known as *access patterns*) still matter

- Access Pattern:
  - How is a file read from (or written to) disk?
    - Random (irregular) access pattern [6,1,8,3,2,4,7,5]
      - Approximates file fragmentation again!
      - Random access is bad for performance!



Data Access Patterns Matter

- Assume a file's blocks can be arranged contiguously on disk
  - The way we access files (known as *access patterns*) still matter
- Access Pattern:
  - How is a file read from (or written to) disk?
    - Random (irregular) access pattern [6,1,8,3,2,4,7,5]
      - Random access is bad for performance!
      - Approximates file fragmentation again!



Main Takeaways

- Files appear like a sequential entity to humans, but that's an *illusion*
  - Your OS splits the file into blocks
  - Finds a free block on disk to store each chunk
  - Sequential ordering of blocks for a file is **not** guaranteed
    - File may be "fragmented" across disk
    - What does fragmentation do to performance of reading the file?

Topics

- Disks
  - Hardware
  - Understanding Files and Blocks
  - Modeling Access Time
  - Disk Scheduling
- Buffer Management

Modeling Average Data Access Time (DAT)

$$DAT = T_{decode} + T_{seek} + T_{lat} + T_{transfer}$$

- Decoding Overhead (T decode)
  - Disk controller decodes the R/W command, initializes physical movements
- Seek Time (T seek)
  - Time taken to move arm to the desired track.
  - Moving to adjacent tracks = **0 cost.**
- Rotational Latency (T lat)
  - Spin the desired data under the arm. Reciprocal of RPM (rotations per minute)
  - **On average,** you need **(full cycle/2)** rotations per block if block are not contiguous
- Transfer Time (T transfer)
  - Time taken to send data payload to/from host machine

Example

$$DAT = T_{decode} + T_{seek} + T_{lat} + T_{transfer}$$

- Estimate the amount it time it would take to read a 1 GB file off a disk with the following characteristics
  - Sector (and block) size = 4 KB; Track size = 2 GB
  - Decode overhead = *0.1 ms* per block fetched
  - 7200 rotations/minute (RPM)
  - 4ms seek time per sector (if next block is on a different track)
  - Disk is connected to host via USB 3.0 (= 5/8 GB/s)
  - Assume sequential access pattern
- File stored contiguously vs File stored randomly on disk (fragmented)

Seek Time + Latency Dominate

$$DAT = T_{decode} + T_{seek} + T_{lat} + T_{transfer}$$

- If file is contiguously arranged on disk and access is sequential
  - DAT = 26214.4ms + **4ms** + **4.167ms** + 1600ms
  - DAT = 27.82 sec
- If file is fragmented on disk and/or access is irregular
  - DAT = 26214.4ms + **2^20ms** + **2^18*4.167ms** + 1600ms
  - DAT = 2168.7 sec = ~36 minutes

Accelerating (Magnetic) Disk-Access Times

- Try to keep blocks pertaining to same file close together on disk
  - Contiguously if possible (to accelerate sequential file access)
  - *Whose job? Operating System (file system)*

- Access disk data in a sequential pattern
  - Random access patterns approximate block-level file fragmentation
  - e.g., `nextLine()` is good for performance
  - *Whose job? Programmer*

Accelerating (Magnetic) Disk-Access Times (2)

- Speed up disk rotations
  - Can we reduce latency?
  - *Whose job? Computer engineers*
    - *(But 7200 RPM is already near physical limitations)*

- Dispatch (schedule) disk operations in a smarter way
  - Can we reduce seek time?
  - *Whose job? DB and/or OS* ***(Next)***

Topics

- Disks
  - Hardware
  - Understanding Files and Blocks
  - Modeling Access Time
  - Disk Scheduling
- Buffer Management

Copying a File

Disk Scheduling

- ***Problem:*** *Seek time is a dominant term in DAT.*

- *Disk Scheduling*
  - Given multiple outstanding disk block requests, in what order should we dispatch them to the disk?

- **Goals:** We want a schedule that
  - Minimizes total latency (delay) for *small* transfers
    - e.g., a selective database query that only returns a couple records from a table
  - Maximizes throughput (bytes per unit time) for *large* transfers
    - e.g., returning all records in a large database table

FCFS Policy

First-Come-First-Served Policy

- Schedule each disk I/O operation in the order it arrives

- Pros
  - It's fair! Requests honored in the order in which they arrive
  - Easy to manage pending requests with a queue (fast!)
- Cons
  - Disk-head locality is not taken into consideration
  - Wide alternating arm swings are possible.

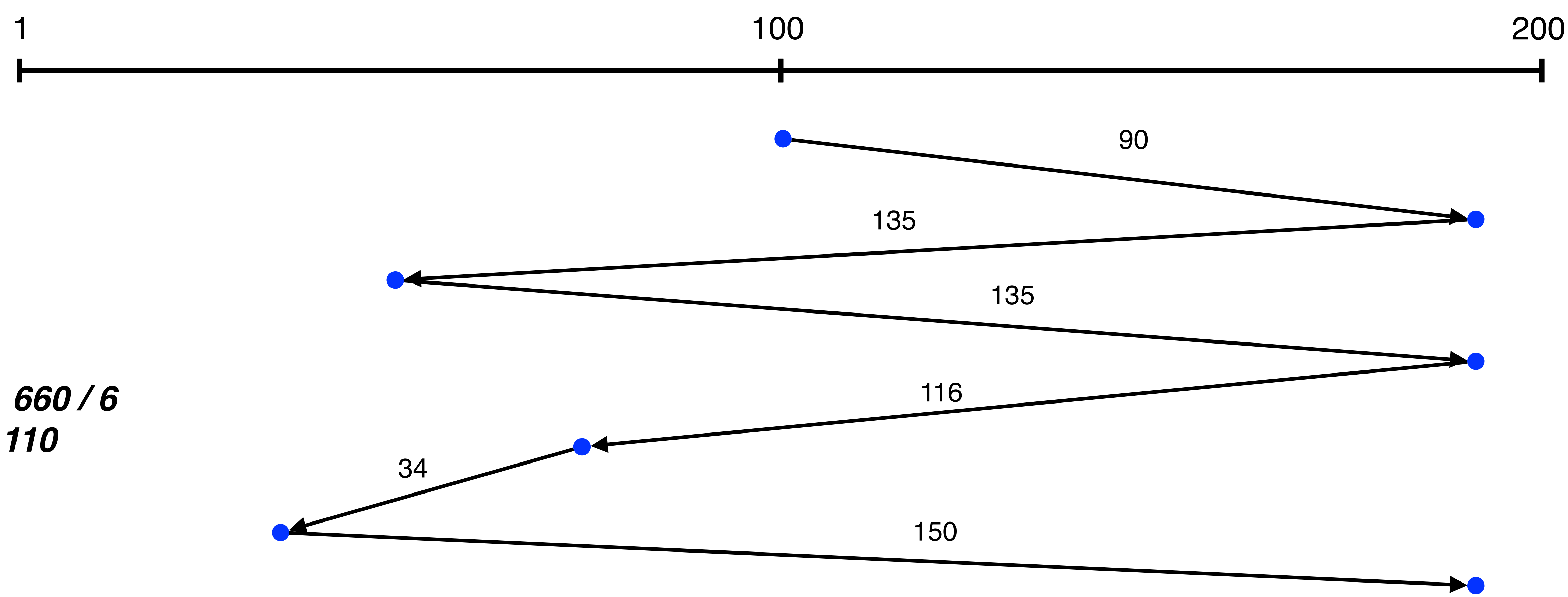
FCFS Example

- Disk head currently on track **100**
- Outstanding requests for I/O to blocks on tracks:
  - **190**, **55**, **190**, **74**, **40**, **190**
  - (Different colors = concurrent DB transactions)
- Let's define a simple metric to compare policies
  - *Average Seek Length per Block*
    - *(Tracks traversed / number of blocks fetched) in the schedule*
    - *Approximates seek time*

FCFS Example

UNIVERSITY of PUGET SOUND

- Disk head currently on track **100**
- Outstanding requests for I/O to blocks on tracks:
  - **190**, **55**, **190**, **74**, **40**, **190**
  - (Different colors = concurrent DB transactions)



ASL = 660 / 6
= 110

SSTF Policy

- Problem with FCFS is that disk-head location is not considered

Shortest-Seek-Time-First (SSTF) Policy

1. Keep track of the current head position
2. Always issue the I/O operation that requests a track nearest to the head
3. If there's a tie, use FCFS or flip a coin

SSTF Example

- Disk head currently on track **100**
- Outstanding requests for I/O to blocks on tracks:
  - **190**, **55**, **190**, **74**, **40**, **190**
  - (Same example as before)
- Evaluation
  - Average seek length = ?
    - *Is SSTF **optimal** for minimizing average seek length?*
  - Potential problems?

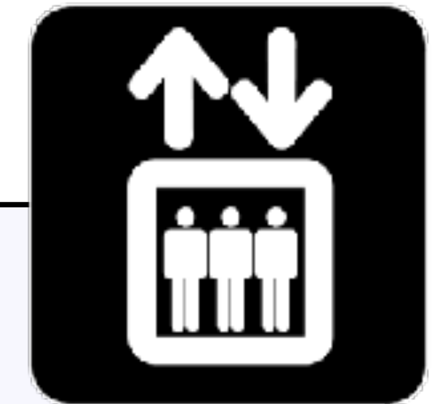
SSTF Example

- Disk head currently on track **100**
- Outstanding requests for I/O to blocks on tracks:
  - **190**, **55**, **190**, **74**, **40**, **190**
  - (Same example as before)

- Evaluation
  - Average seek length = ?
    - *Is SSTF* ***optimal*** *for minimizing average seek length?*
  - Potential problems?
    - *Starvation* is possible for outer tracks as requests continually arrive? (Why?)

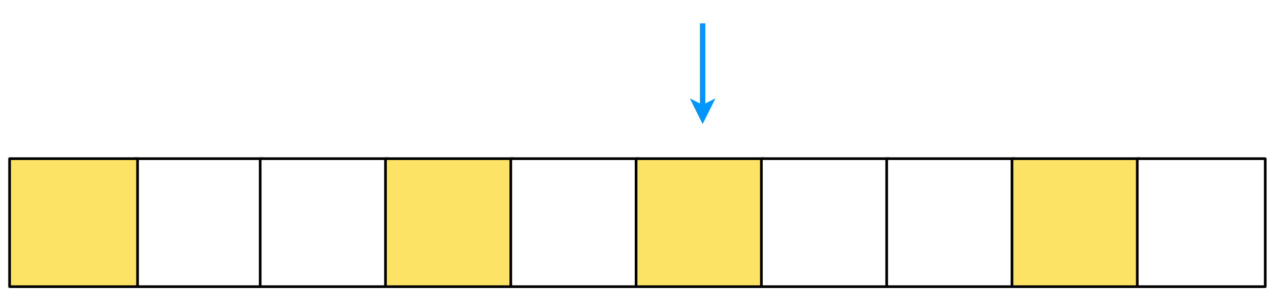
LOOK Policy (Also called Elevator)

- Starvation is a problem!



LOOK Policy

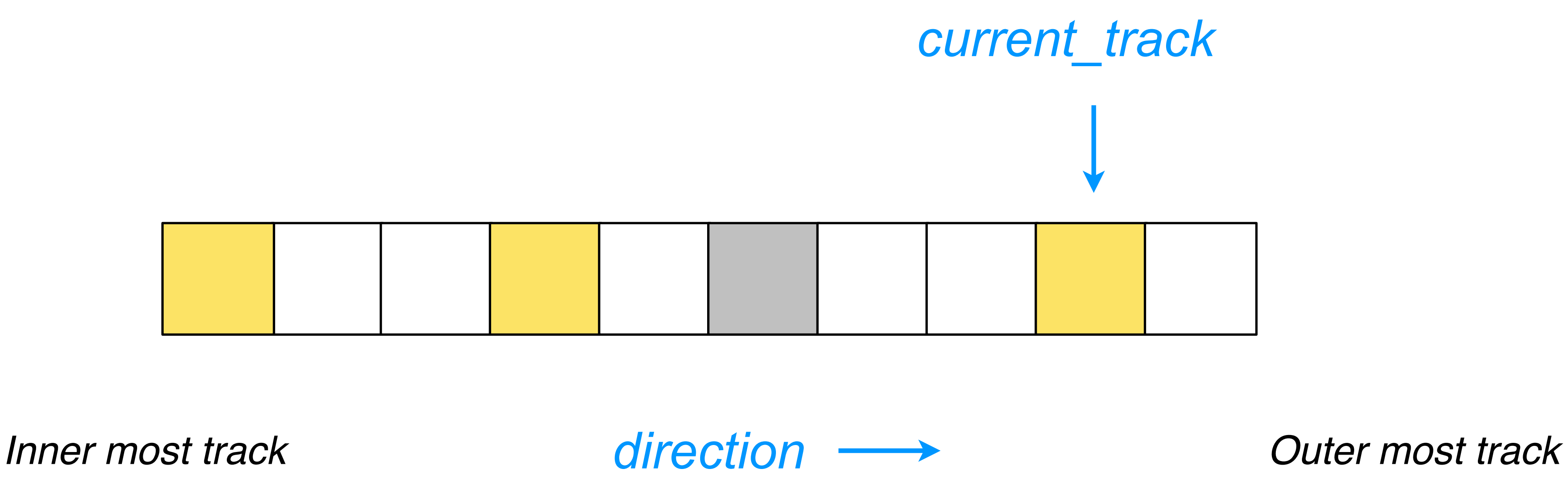
1. Disk-head has *direction* and *current_track*
2. Sort requests in order of track number according to *direction*
3. If no more I/Os to service in *direction*, reverse direction



LOOK Policy

LOOK Policy

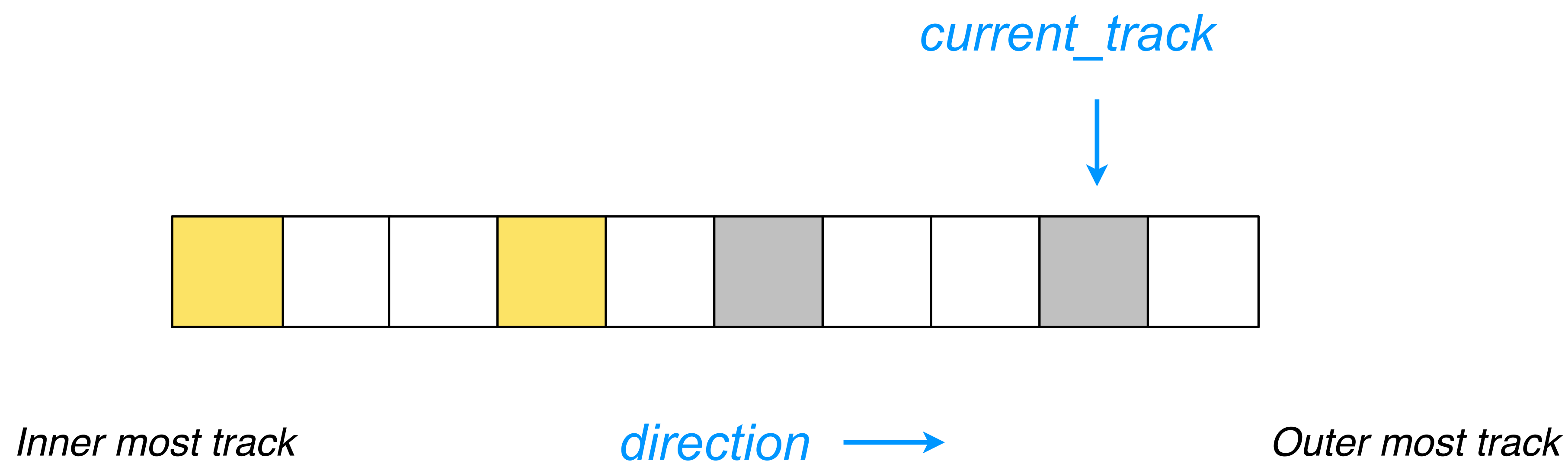
1. Disk-head has *direction* and *current_track*
2. Sort I/O in order of track number according to *direction*
3. If no more I/Os to service in *direction*, reverse direction



LOOK Policy

LOOK Policy

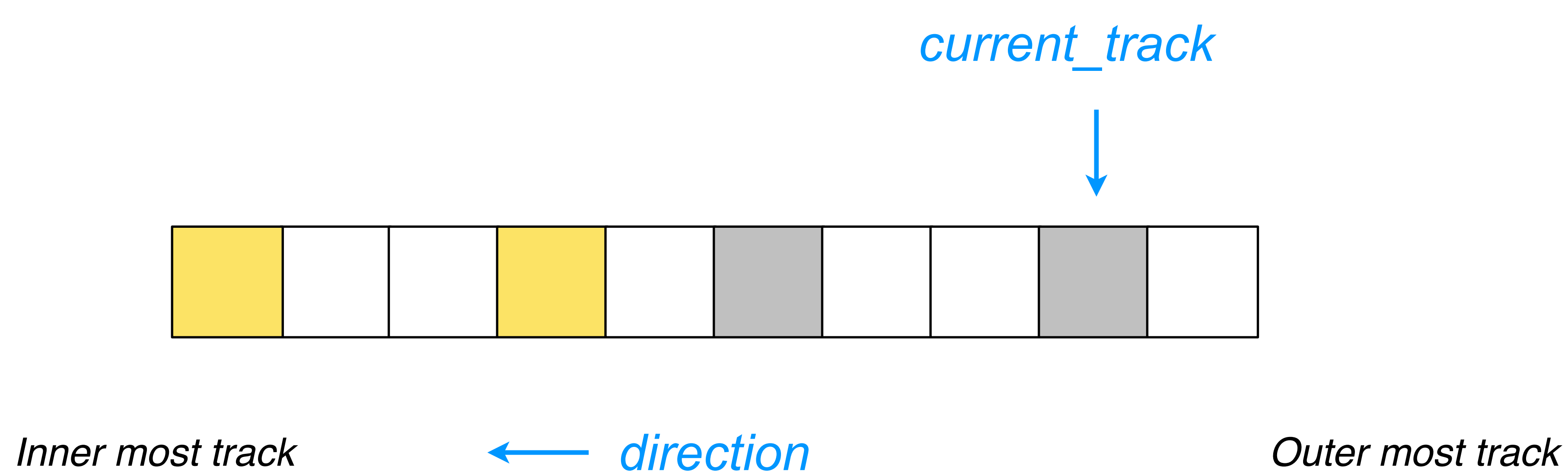
1. Disk-head has *direction* and *current_track*
2. Sort I/O in order of track number according to *direction*
3. If no more I/Os to service in *direction*, reverse direction

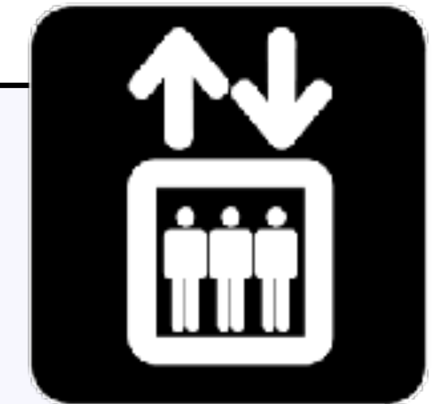


LOOK Policy

LOOK Policy

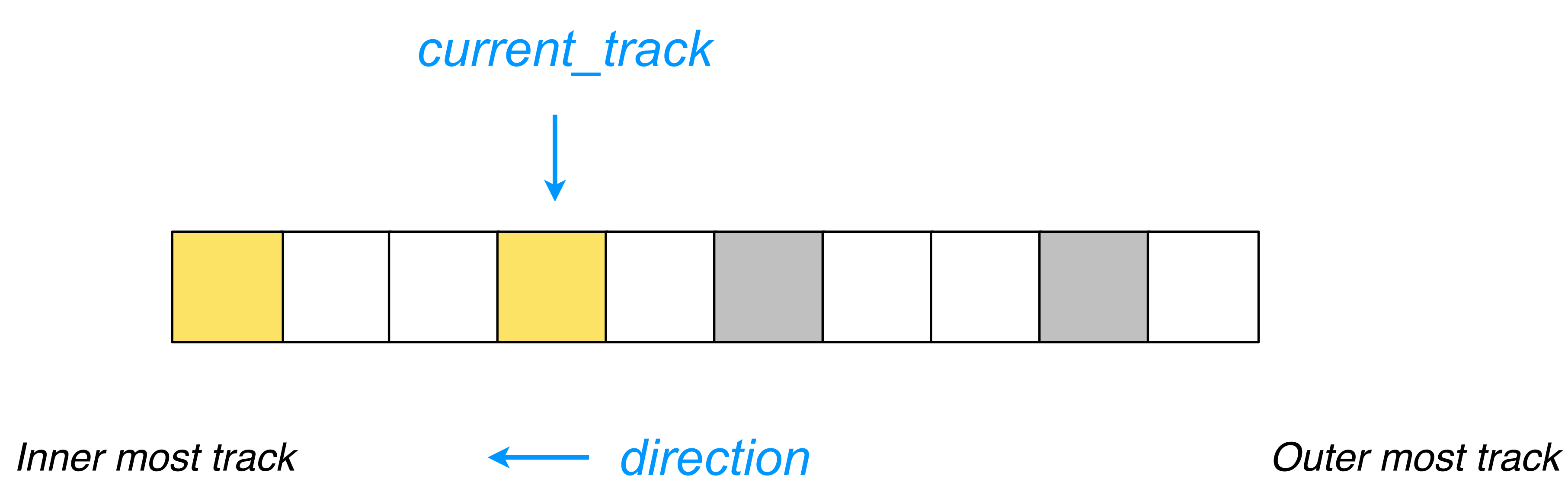
1. Disk-head has *direction* and *current_track*
2. Sort I/O in order of track number according to *direction*
3. If no more I/Os to service in *direction*, reverse direction





LOOK Policy

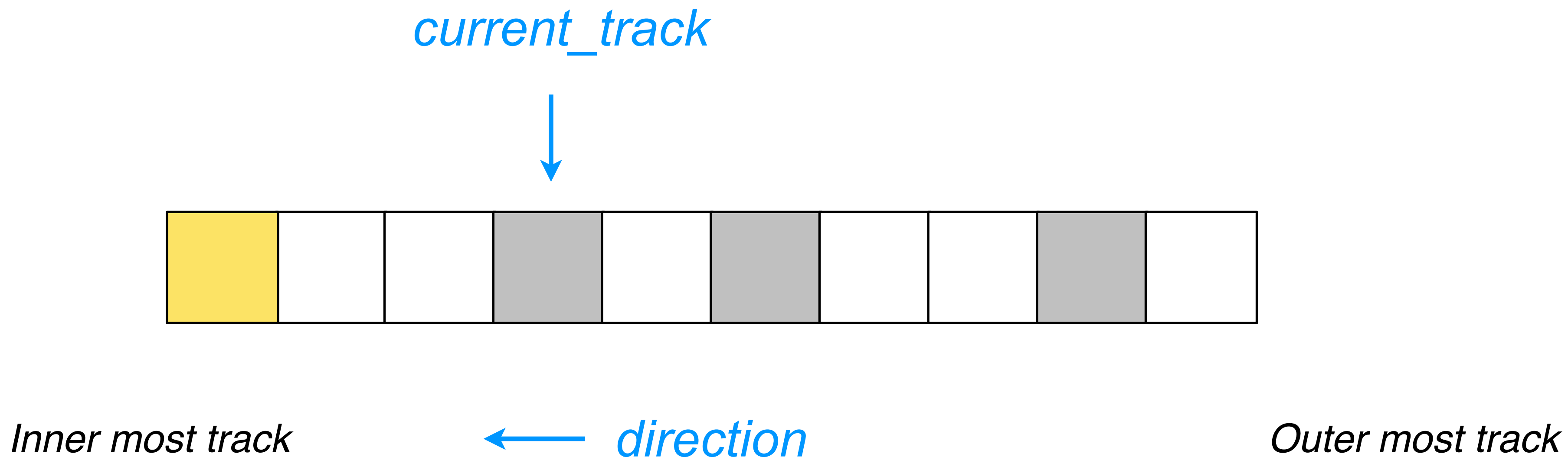
1. Disk-head has *direction* and *current_track*
2. Sort I/O in order of track number according to *direction*
3. If no more I/Os to service in *direction*, reverse direction

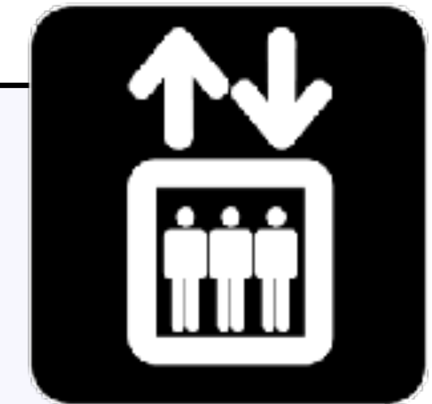


LOOK Policy

LOOK Policy

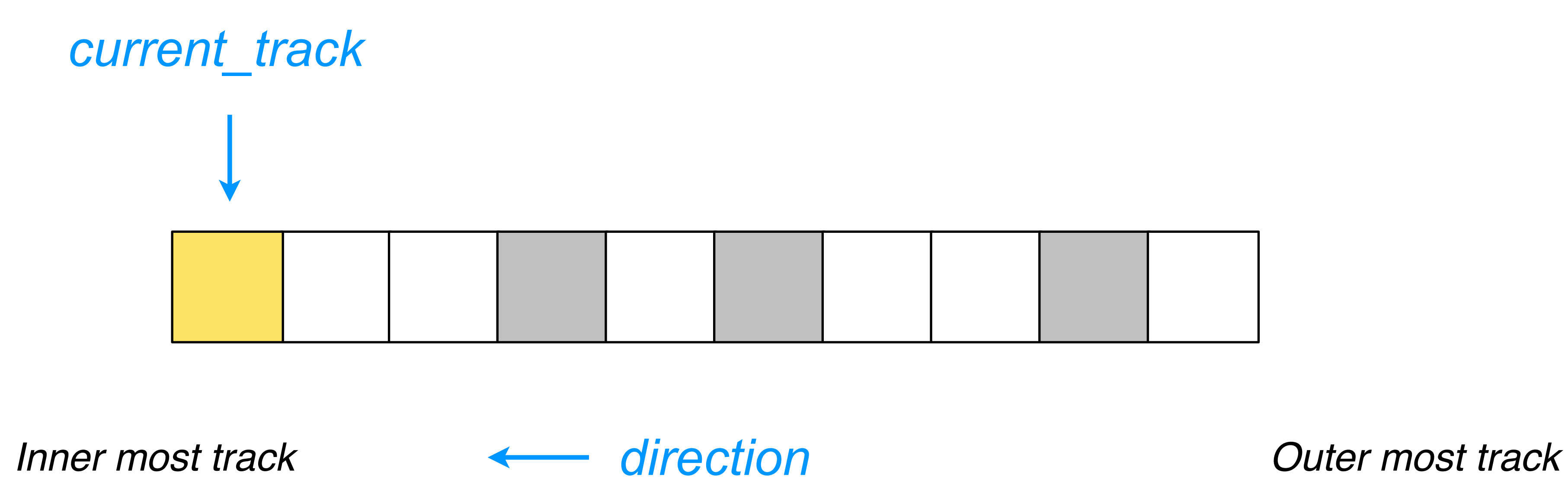
1. Disk-head has *direction* and *current_track*
2. Sort I/O in order of track number according to *direction*
3. If no more I/Os to service in *direction*, reverse direction





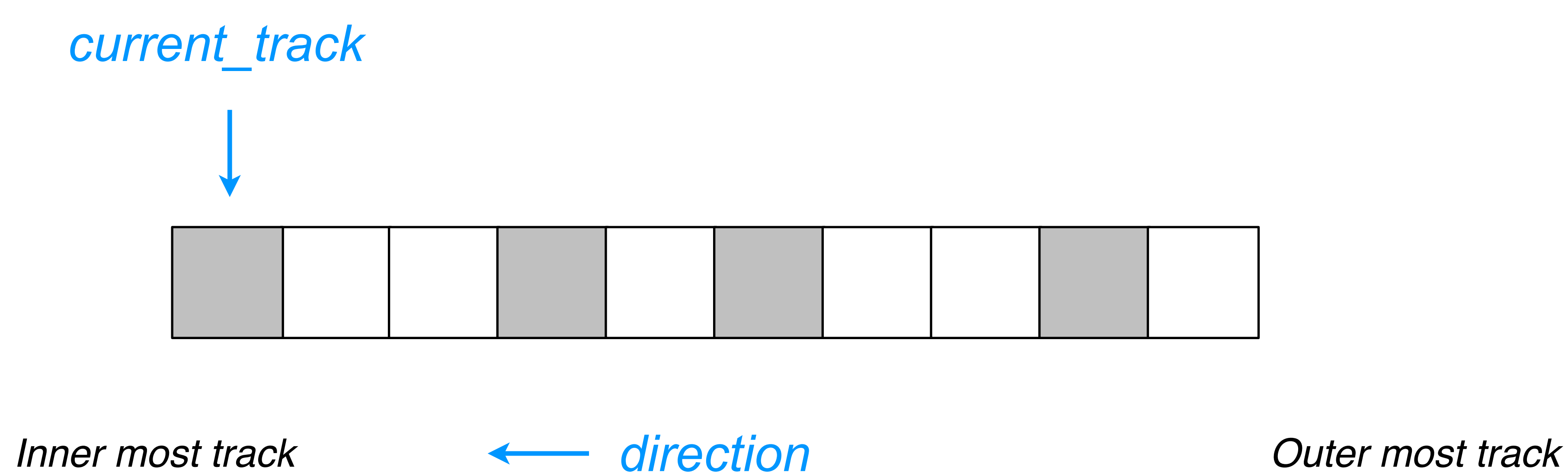
LOOK Policy

1. Disk-head has *direction* and *current_track*
2. Sort I/O in order of track number according to *direction*
3. If no more I/Os to service in *direction*, reverse direction



LOOK Policy

1. Disk-head has *direction* and *current_track*
2. Sort I/O in order of track number according to *direction*
3. If no more I/Os to service in *direction*, reverse direction



LOOK Example

- Disk head currently on track **53**
- Outstanding requests for I/O to sectors on cylinders:
  - 98, 183, 37, 122, 14, 124
- Evaluation
  - Average seek length = ?
  - Starvation reduced, but still possible

Circular LOOK Policy

- How to guarantee all tracks are visited periodically?
  - Think typewriter
  - (Not implemented in practice -- too wasteful)

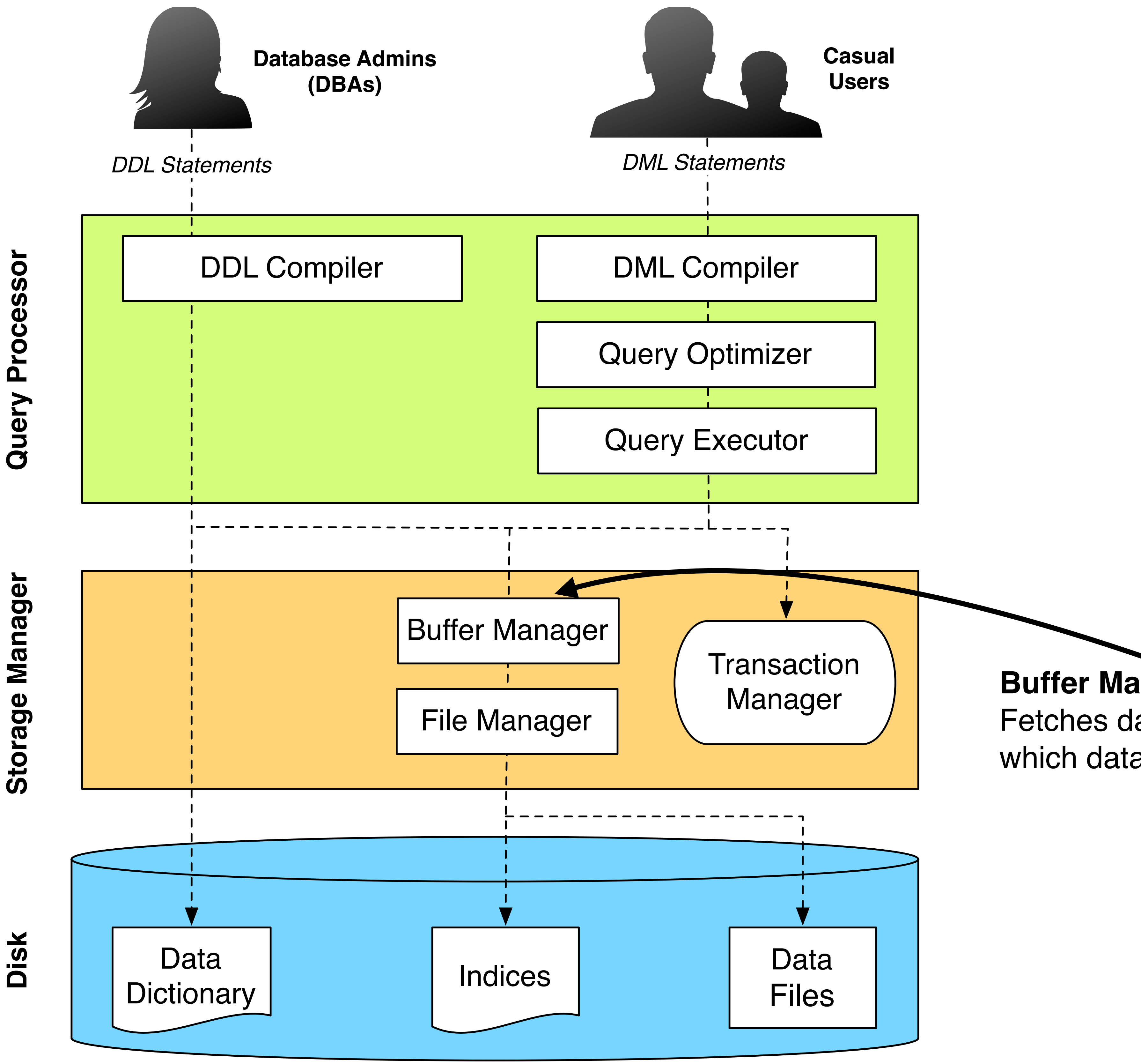
Circular LOOK Policy

1. Just like LOOK, but head moves in only one *direction*
2. If no more I/Os to service in *direction*, move head to track #**0** (Ignoring any requests on the way back to track #**0**)

Topics

- Disks
  - Hardware
  - Understanding Files and Blocks
  - Modeling Access Time
  - Disk Scheduling
- Buffer Management

Database Architecture



Buffer Manager
Fetches data from disk into memory, deciding which data to cache in memory.

Motivation: Buffer Management

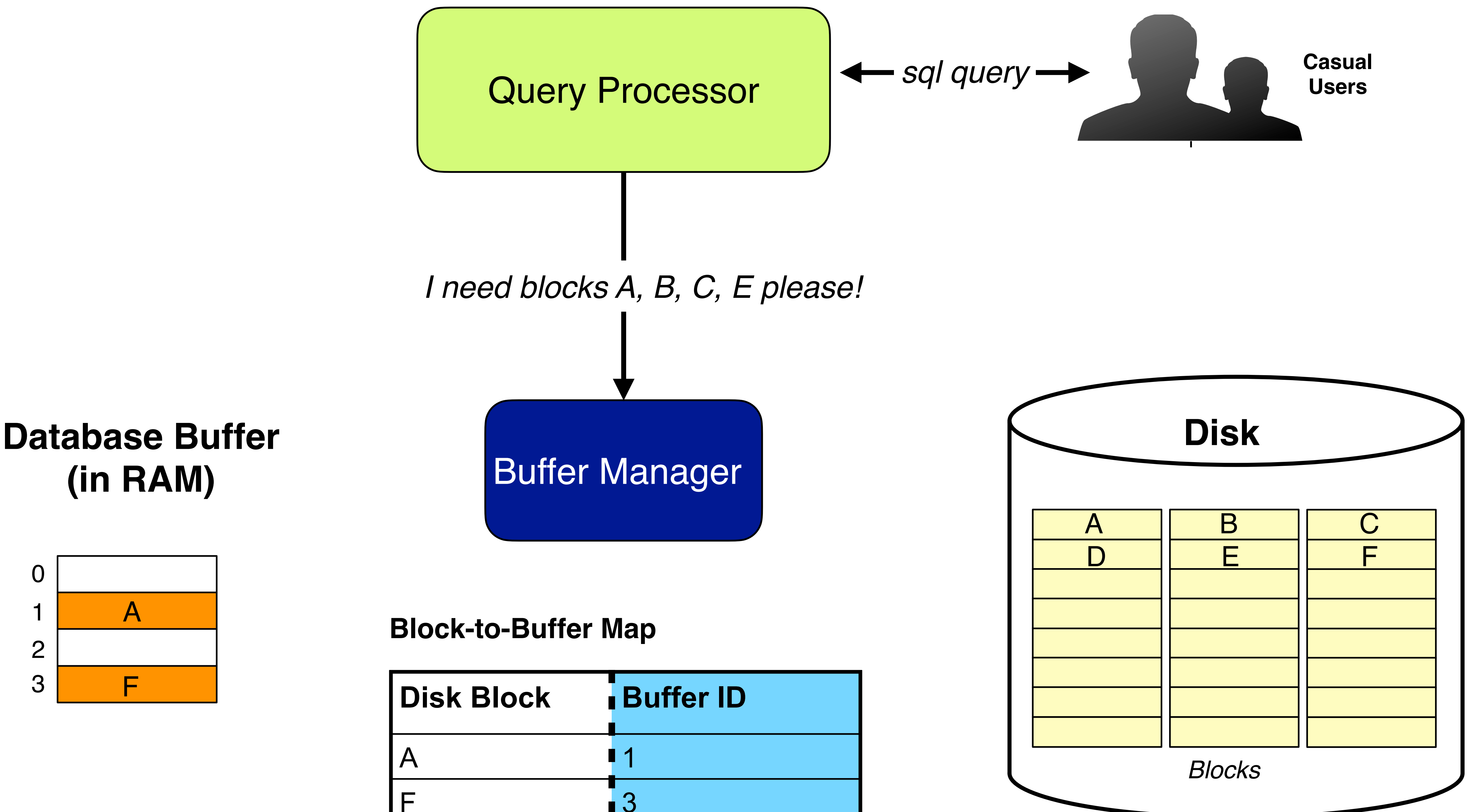
- **Problem:**
  - Database size (on disk) is usually larger than available memory!
    - Terabytes (Disk) compared to GBs (RAM)

      ~3 orders of magnitude difference!

- **Goal:** Minimize number of block transfers between disk and memory while processing queries.

- A *database buffer* is a segment of memory that the DB uses to *cache* some disk blocks for later access.

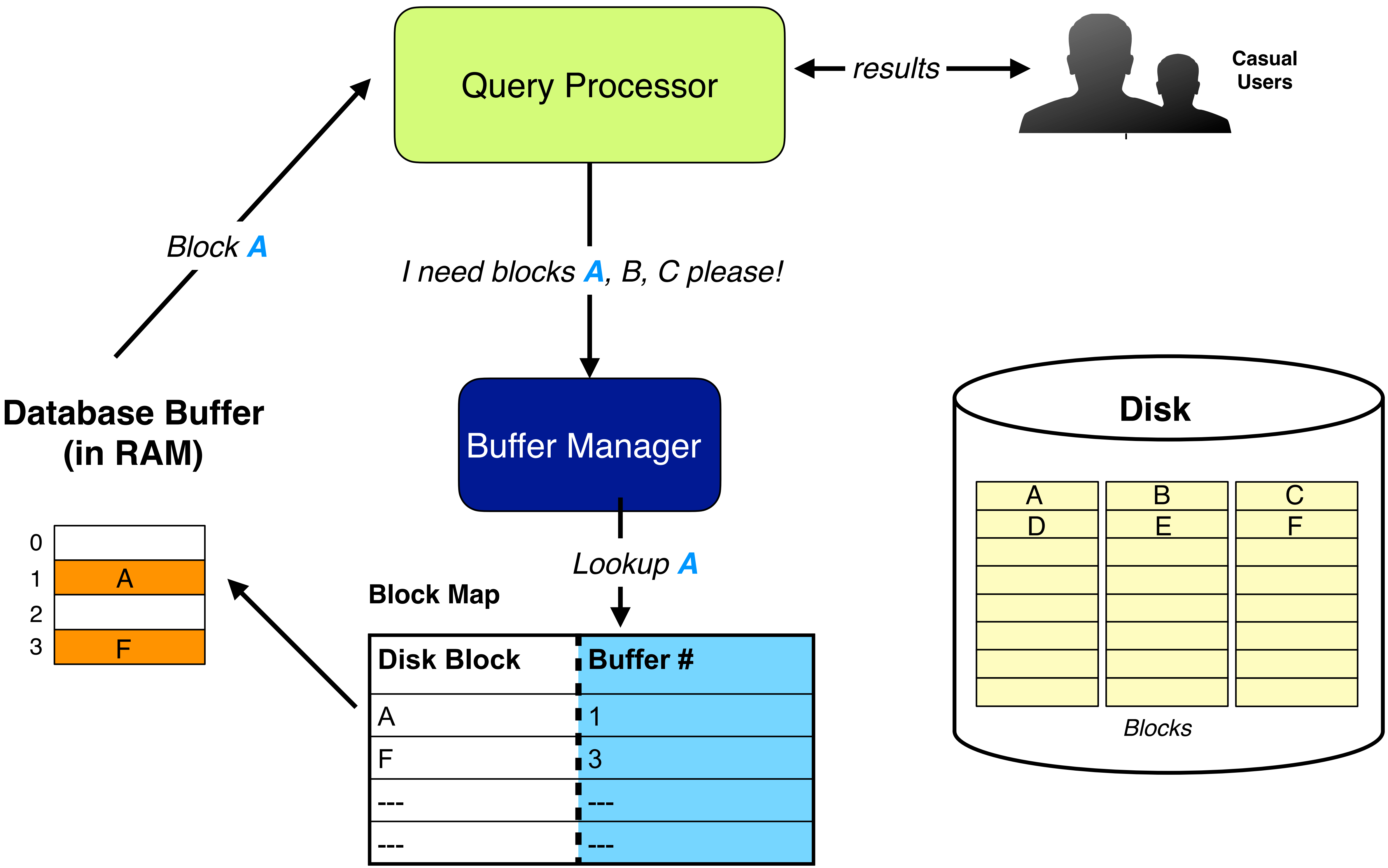
Buffer Manager



Block-to-Buffer Map

Disk Block	Buffer ID
A	1
F	3
---	---
---	---

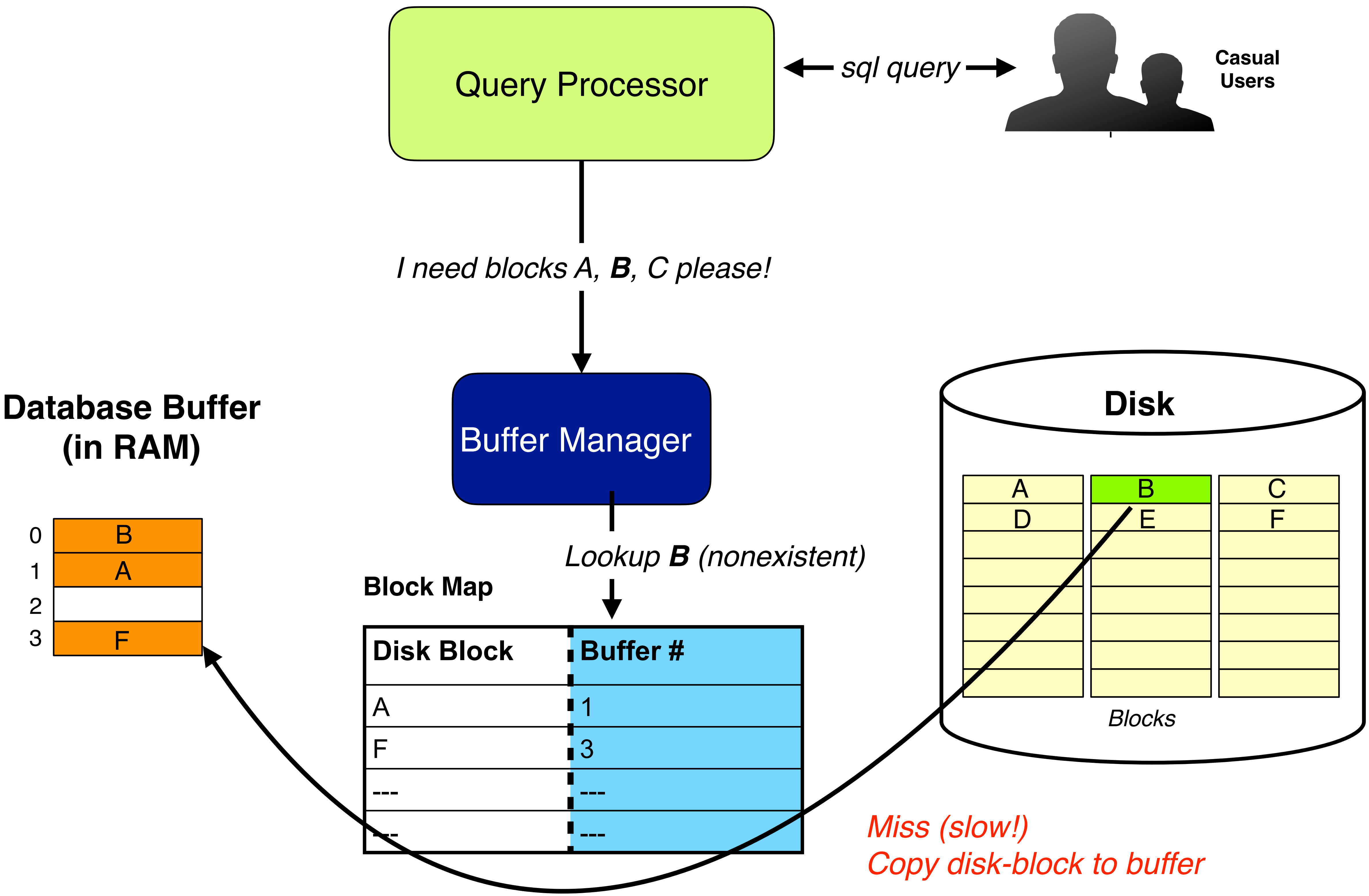
Buffer Manager



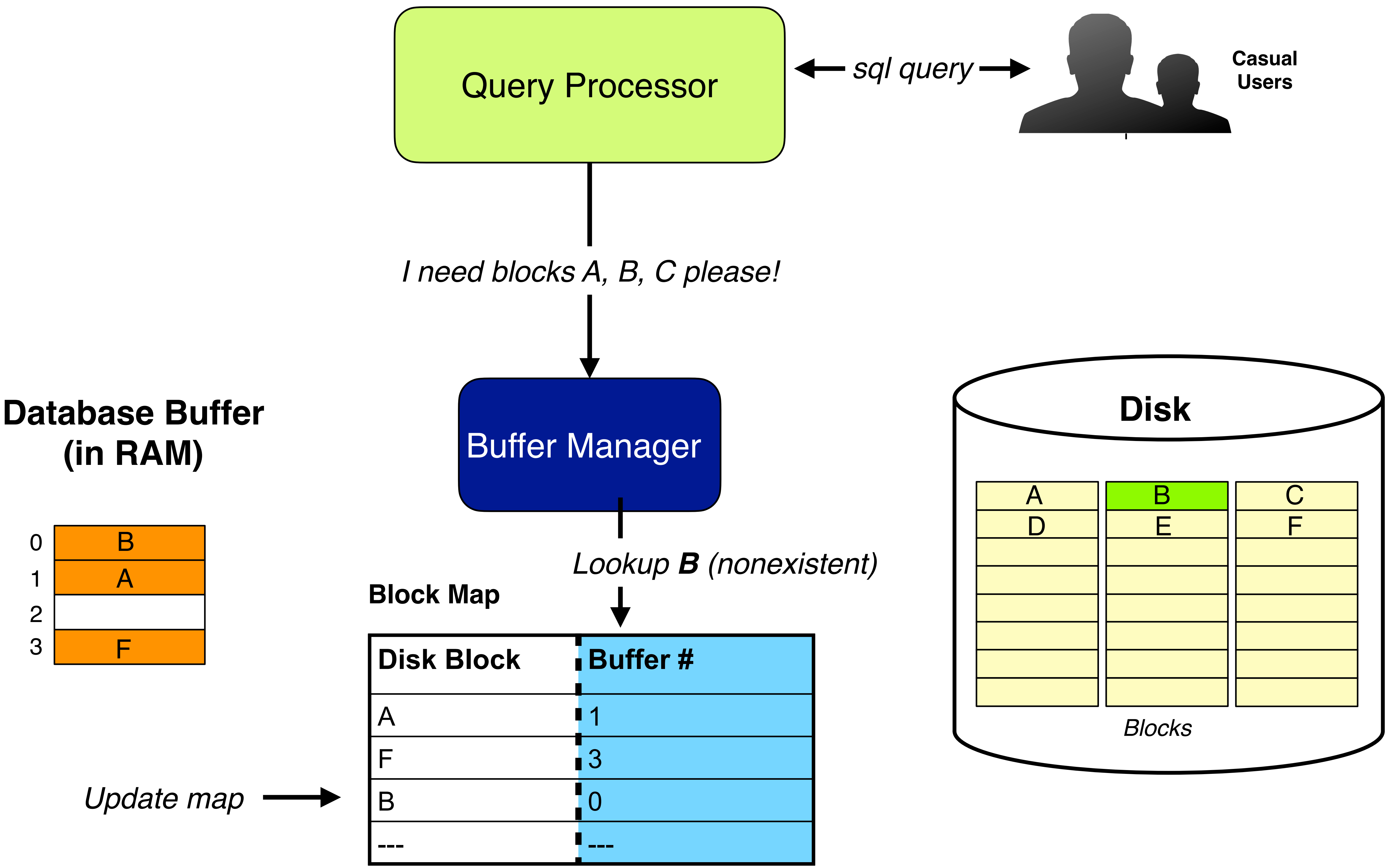
Block Map

Disk Block	Buffer #
A	1
F	3
---	---
---	---

Buffer Manager

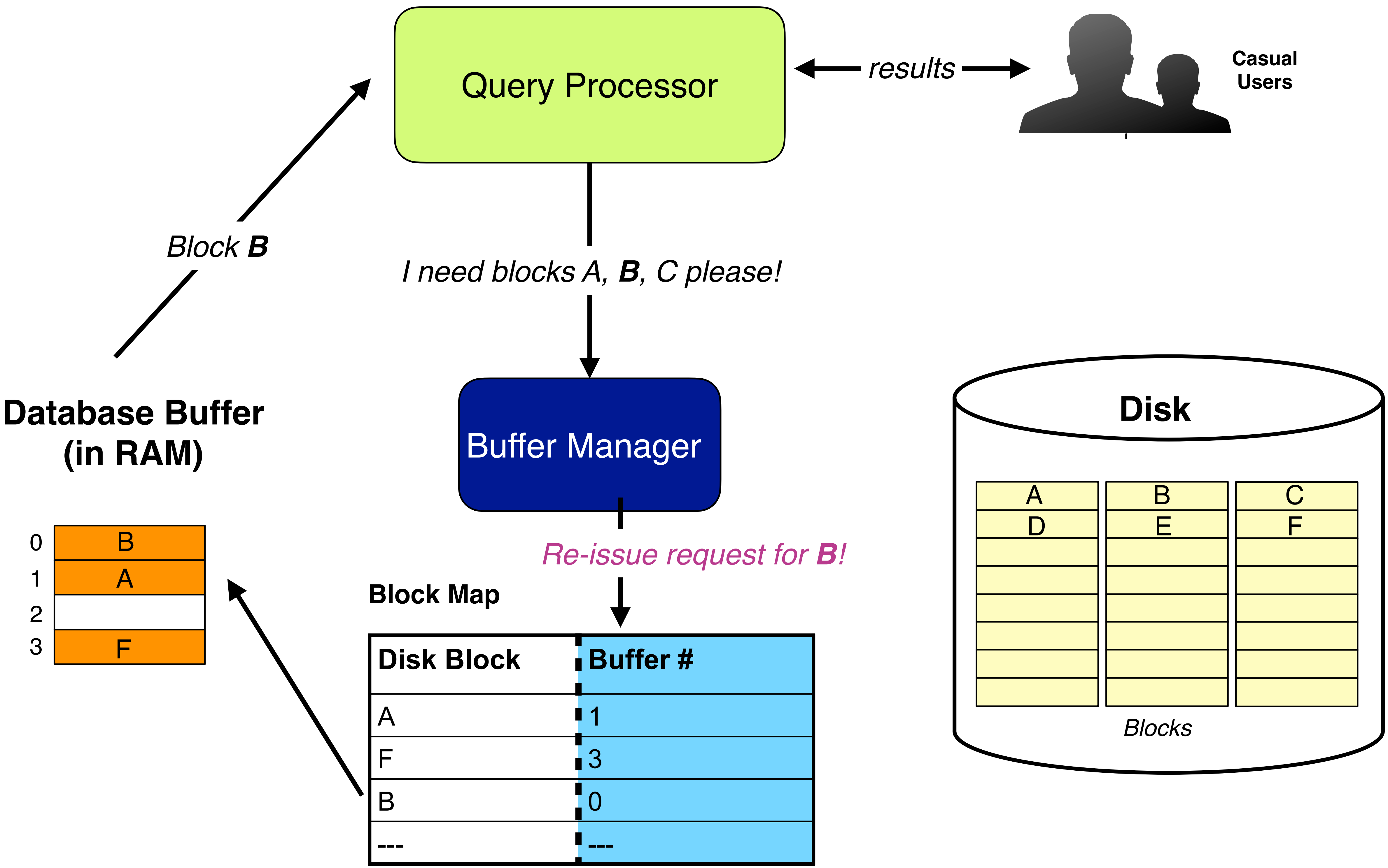


Buffer Manager



Disk Block	Buffer #
A	1
F	3
B	0
---	---

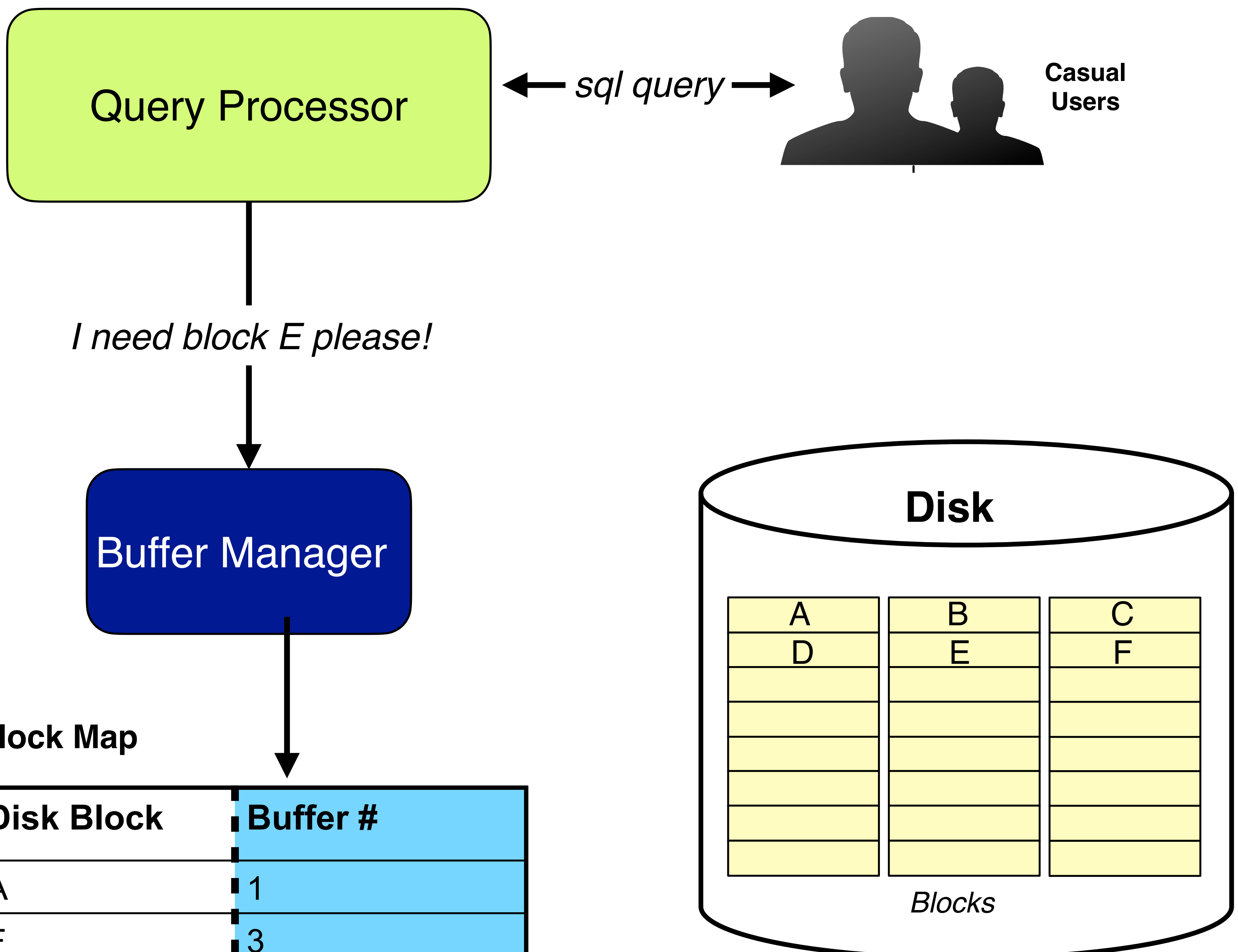
Buffer Manager



Block Map

Disk Block	Buffer #
A	1
F	3
B	0
---	---

New Problem: What If Buffer Is Full?



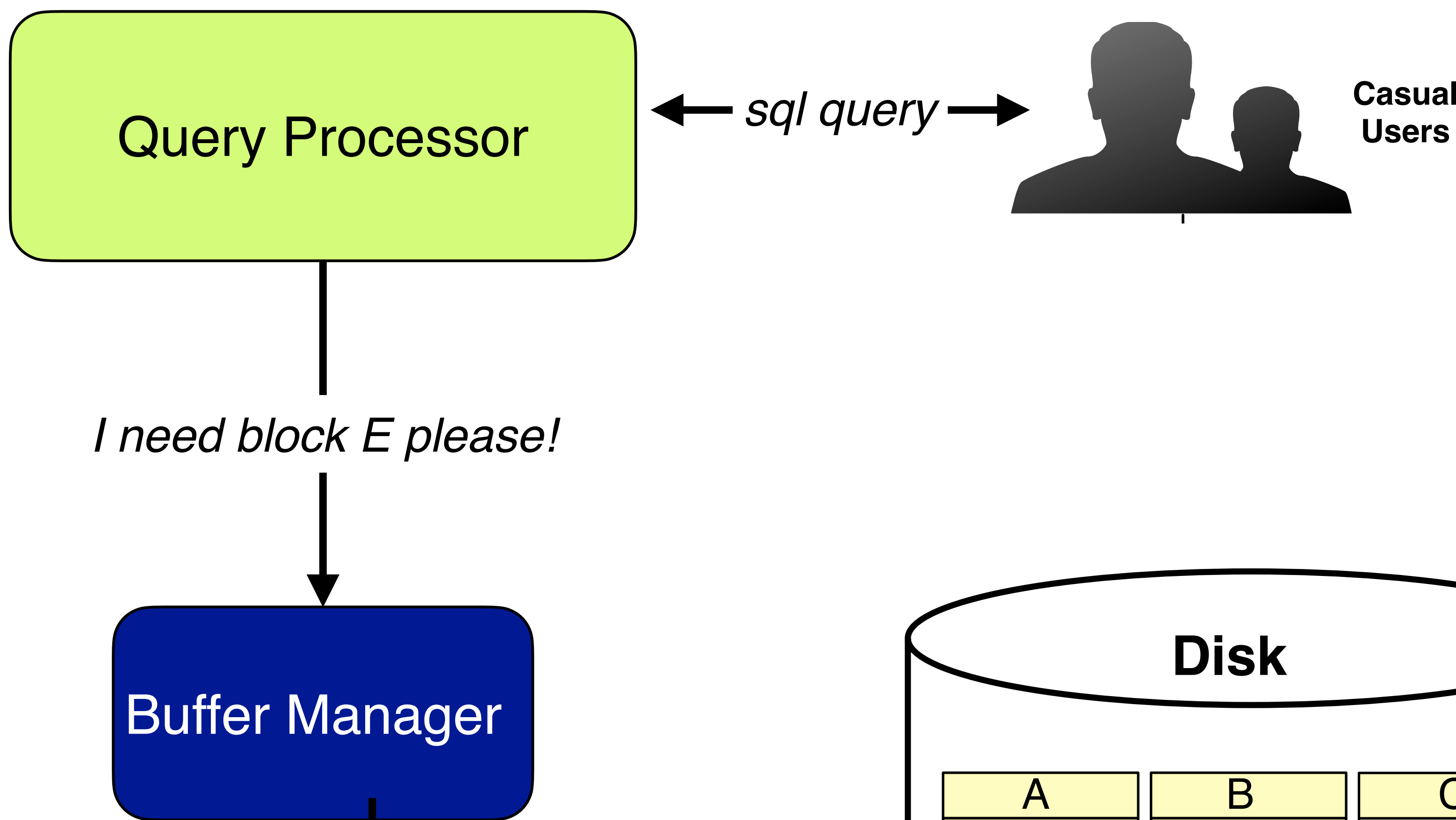
Database Buffer (in RAM)

0	B
1	A
2	C
3	F

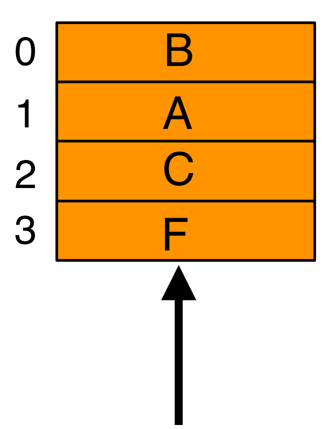
Block Map

Disk Block	Buffer #
A	1
F	3
B	0
C	2

New Problem: What If Buffer Is Full?



Database Buffer (in RAM)

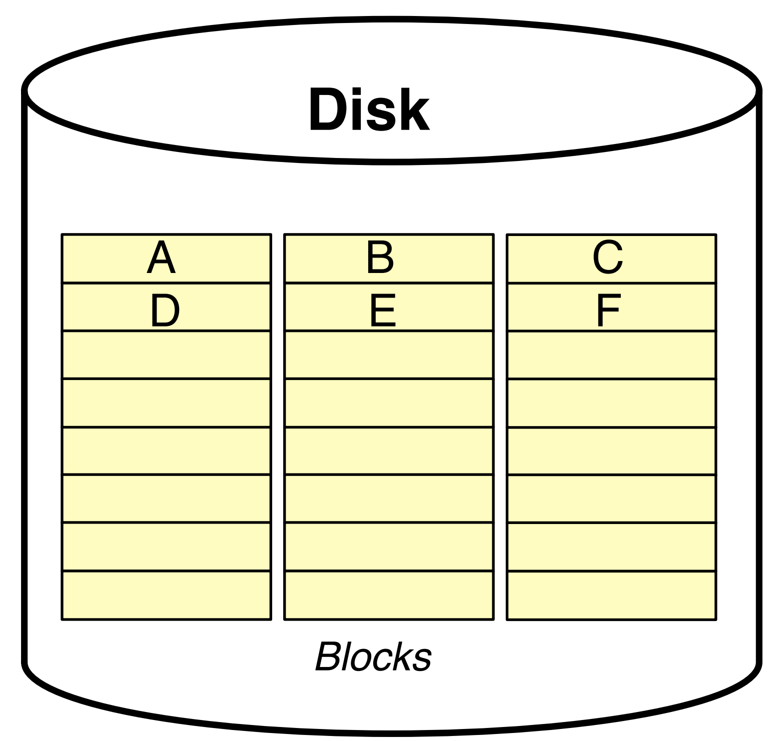


Need to kick out a block, but which one?

(Replacement Policy)

Block Map

Disk Block	Buffer #
A	1
F	3
B	0
C	2



Common Block-Replacement Policies

- What does Buffer Manager do when buffer is full?

- *First In First Out (FIFO) Policy*
  - The *oldest* block allocated in the buffer gets kicked out.
  - Pros: Fast; Simple to implement using queue
  - Cons: An *old* block doesn't mean it's hardly-ever used!

- *Least Recently Used (LRU) Policy*
  - The block that was used *farthest in the past* gets kicked out.
  - Pros: Exploits temporal locality
  - Cons: Harder to implement (uses priority queue)

Example

- Assume we only have *2 blocks* in the database buffer.
- Two tables: Book(BNO, title, author) and Publisher(PNO, BNO, Name)
    - 100 Books tuples can be stored in a block
    - 50 Publisher tuples can be stored in a block

```
Book
--------------------
BNO | title | author
```

Block B1
```
book1
...
book100
```

Block B2
```
book101
...
book200
```

Block B3
```
book201
...
book300
```

```
Publisher
--------------------
PNO | BNO | Name
```

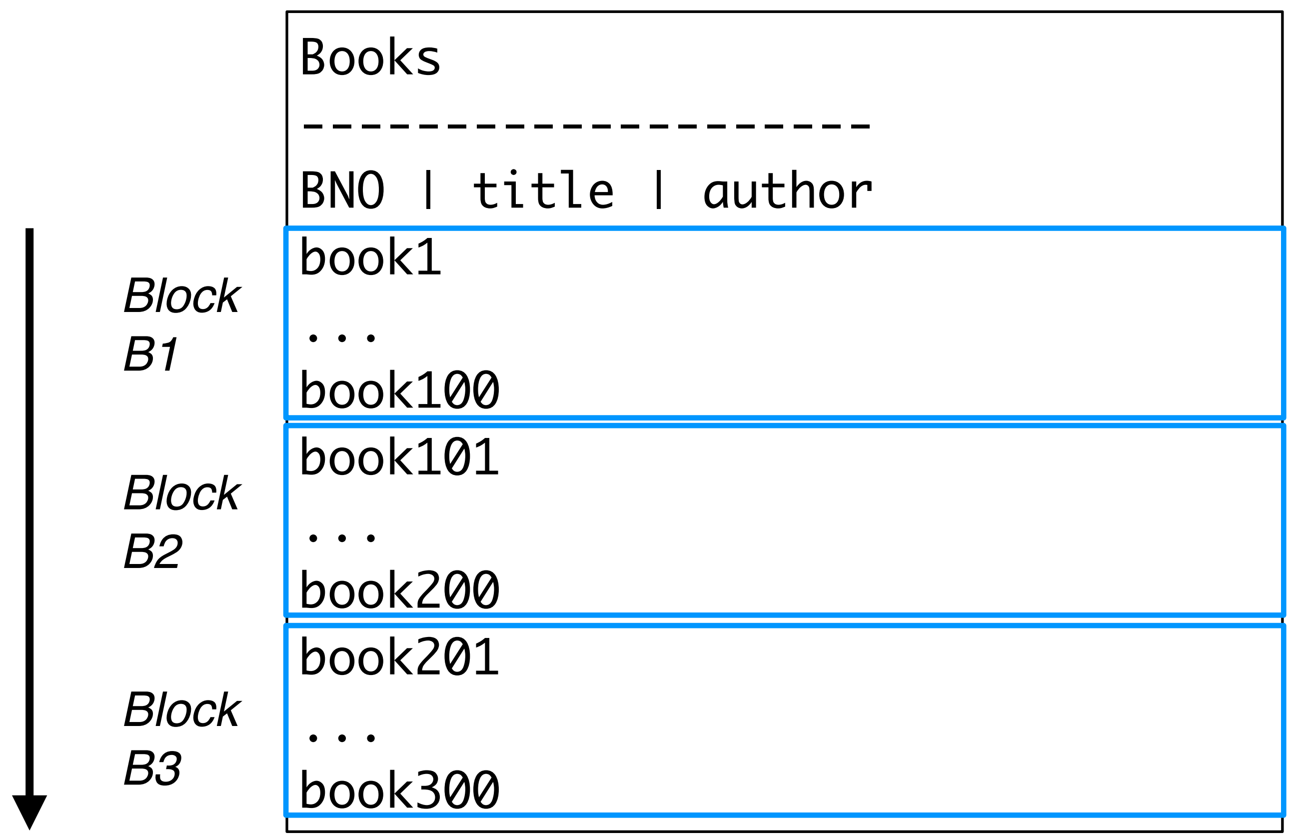
Block P1
```
pub1
...
pub50
```

Block P2
```
pub51
...
pub100
```

Example (Cont.)

- Let's run a simple select query: `select * from Books;`

- Access Pattern:



FIFO Replacement Policy

- Database buffer allocates **two** blocks in RAM
  - 3 file blocks for the Books relation: B1, B2, B3
  - Access Pattern of select_query(): B1 (50x), B2 (50x), B3 (50x)

Time →

Access Pattern	**B1**	**...**	**B1**	**B2**	**...**	**B2**	**B3**	**...**	**B3**
Buffer Slot 1	B1	B1	B1	B1	B1	B1	B3	B3	B3
Buffer Slot 2				B2	B2	B2	B2	B2	B2

50 x

50 x

50 x

Number of misses to run query = 3

LRU Replacement

- Database buffer allocates **two** blocks in RAM
  - 3 file blocks for the Books relation: B1, B2, B3
  - Access Pattern of select_query(): B1 (50x), B2 (50x), B3 (50x)

Time →

Access Pattern	**B1**	**...**	**B1**	**B2**	**...**	**B2**	**B3**	**...**	**B3**
Buffer Slot 1	B1	B1	B1	B1	B1	B1	B3	B3	B3
Buffer Slot 2				B2	B2	B2	B2	B2	B2

50 x 50 x 50 x

Number of misses = 3 (same as FIFO for this query)

Access Pattern for a Join

```
Books
--------------------
BNO | title | author
book1
book2
...
book100
book101
...
book200
book201
...
book300
```

Block B1

Join with book1

. . .

```
Publisher
--------------------
PNO | BNO | Name
pub1
...
pub50
pub51
...
pub100
```

Block P1

Block P2

Tuple generated:	*(book1,pub1)*	*(book1,pub2)*	...	*(book1 , pub50)*	*(book1 , pub51)*	...	*(book1 , pub100)*
Block:	B1 P1	B1 P1	...	B1 P1	B1 P2	...	B1 P2

Access Pattern for a Join

UNIVERSITY of PUGET SOUND

```
Books
--------------------
BNO | title | author
book1
book2
...
book100
book101
...
book200
book201
...
book300
```

Block B1

Join with book2

```
Publisher
--------------------
PNO | BNO | Name
pub1
...
pub50
pub51
...
pub100
```

Block P1

Block P2

Tuple:	*(book1,pub1)*	*(book1,pub2)*	...	*(book1 , pub50)*	*(book1 , pub51)*	...	*(book1 , pub100)*
Block:	B1 P1	B1 P1		B1 P1	B1 P2		B1 P2
Tuple:	*(book2,pub1)*	*(book2,pub2)*	...	*(book2 , pub50)*	*(book2 , pub51)*	...	*(book2 , pub100)*
Block:	B1 P1	B1 P1		B1 P1	B1 P2		B1 P2

...

Tuple:	*(book100,pub1)*	...	*(book100 , pub50)*	*(book100, pub51)*	...	*(book100 , pub100)*
Block:	B1 P1		B1 P1	B1 P2		B1 P2

Access Pattern for a Join

```
Books
--------------------
BNO | title | author
book1
book2
...
book100
book101
...
book200
book201
...
book300
```

Block B1 (book1 … book100)

Join with book2

```
Publisher
--------------------
PNO | BNO | Name
pub1
...
pub50
pub51
...
pub100
```

Block P1 (pub1 … pub50)

Block P2 (pub51 … pub100)

100 times for block B1

Tuple:	*(book1,pub1)*	*(book1,pub2)*	...	*(book1 , pub50)*	*(book1 , pub51)*	...	*(book1 , pub100)*
Block:	B1 P1	B1 P1		B1 P1	B1 <u>P2</u>		B1 <u>P2</u>
Tuple:	*(book2,pub1)*	*(book2,pub2)*	...	*(book2 , pub50)*	*(book2 , pub51)*	...	*(book2 , pub100)*
Block:	B1 P1	B1 P1		B1 P1	B1 <u>P2</u>		B1 <u>P2</u>
				⋱			
Tuple:	*(book100,pub1)*	...		*(book100 , pub50)*	*(book100, pub51)*	...	*(book100 , pub100)*
Block:	B1 P1			B1 P1	B1 <u>P2</u>		B1 <u>P2</u>

Access Pattern for a Join (Cont.)

Books

BNO | title | author
book1
...
book100

Block B2
book101
...
book200

book201
...
book300

Publisher

PNO | BNO | Name

Block P1
pub1
...
pub50

Block P2
pub51
...
pub100

100 times for block **B2**

Tuple:	*(b101 , p1)*	*(b101 , p2)*	...	*(b101 , p50)*	*(b101 , p51)*	*(b101 , p52)*	...	*(b101 , p100)*
Block:	B2 P1	B2 P1		B2 P1	B2 P2	B2 P2		B2 P2
Tuple:	*(b102 , p1)*	*(b102, p2)*	...	*(b102 , p50)*	*(b102 , p51)*	*(b102 , p52)*	...	*(b102 , p100)*
Block:	B2 P1	B2 P1		B2 P1	B2 P2	B2 P2		B2 P2
...								
Tuple:	*(b200 , p1)*	*(b200 , p2)*	...	*(b200 , p50)*	*(b200 , p51)*	*(b200 , p52)*	...	*(b200 , p100)*
Block:	B2 P1	B2 P1		B2 P1	B2 P2	B2 P2		B2 P2

Access Pattern for a Join (Cont.)

```
Books
--------------------
BNO | title | author
book1
...
book100
book101
...
book200
book201
...
book300
```

Block B3

```
Publisher
--------------------
PNO | BNO | Name
pub1
...
pub50
pub51
...
pub100
```

Block P1

Block P2

100 times for block **B3**

Tuple:	*(b201 , p1)*	*(b201 , p2)*	...	*(b201 , p50)*	*(b201 , p51)*	*(b201 , p52)*	...	*(b201 , p100)*
Block:	B3 P1	B3 P1		B3 P1	B3 P2	B3 P2		B3 P2
Tuple:	*(b202 , p1)*	*(b202, p2)*	...	*(b202 , p50)*	*(b202 , p51)*	*(b202 , p52)*	...	*(b202 , p100)*
Block:	B3 P1	B3 P1		B3 P1	B3 P2	B3 P2		B3 P2
...								
Tuple:	*(b300 , p1)*	*(b300 , p2)*	...	*(b300 , p50)*	*(b300 , p51)*	*(b300 , p52)*	...	*(b300 , p100)*
Block:	B3 P1	B3 P1		B3 P1	B3 P2	B3 P2		B3 P2

FIFO Replacement Policy

- Database buffer allocates **two** blocks in RAM
  - Access Pattern of natural join(Book, Publisher)

Time →

Access Pattern	**B1**	**P1**	**...**	**B1**	**P2**	**B1**	**...**	**B1**	**P1**	**...**	**B1**	**P2**	**B1**	**...**
Buffer Slot 1	B1	B1	B1	B1	P2	P2	P2	P2	P1	P1	P1	P1	B1	B1
Buffer Slot 2		P1	P1	P1	P1	B1	B1	B1	B1	B1	B1	P2	P2	P2

...

50 x

50 x

50 x

50 x

2 misses in first 50 joins: Must fetch **B1** and **P2** from disk initially.

2 misses
This completes the join of tuple **book1** with all tuples in Publishers

1 miss
Next 50 joins for **book2**

2 misses
Remaining 50 joins for **book2**

LRU Replacement Policy

- Database buffer allocates **two** blocks in RAM
  - Access Pattern of natural join(Book, Publisher)

Time →

Access Pattern	**B1**	**P1**	**...**	**B1**	**P2**	**B1**	**...**	**B1**	**P1**	**...**	**B1**	**P2**	**B1**	**...**
Buffer Slot 1	B1	B1	B1	B1	B1	B1	B1	B1	B1	B1	B1	B1	B1	B1
Buffer Slot 2		P1	P1	P1	P2	P2	P2	P2	P1	P1	P1	P2	P2	P2

...

50 x | 50 x | 50 x | 50 x

2 misses in first 50 joins: Must fetch **B1** and **P2** from disk initially.

1 miss
This completes the join of tuple **book1** with all tuples in Publishers

1 miss
Next 50 joins for **book2**

1 miss
Remaining 50 joins for **book2**

Evaluation of FIFO vs. LRU for a Join

- Full Access pattern
  - (DAT = disk access time per block; MAT = RAM access time per block)

Full Access Pattern for NLJ	No-Buffering Misses	FIFO Misses	LRU Misses
B1, P1 (repeated 50 times) B1, P2 (repeated 50 times) ... B1, P1 (repeated 50 times) B1, P2 (repeated 50 times)	50 + 50 + 50 + ... 50 = 5000 misses	2 + 3 + 3 + ... 3 = 150 misses	2 + 2 + 2 + ... 2 = 100 misses
B2, P1 (repeated 50 times) B2, P2 (repeated 50 times) ... B2, P1 (repeated 50 times) B2, P2 (repeated 50 times)	50 + 50 + 50 + ... 50 = 5000 misses	2 + 3 + 3 + ... 3 = 150 misses	2 + 2 + 2 + ... 2 = 100 misses
B3, P1 (repeated 50 times) B3, P2 (repeated 50 times) ... B3, P1 (repeated 50 times) B3, P2 (repeated 50 times)	50 + 50 + 50 + ... 50 = 5000 misses	2 + 3 + 3 + ... 3 = 150 misses	2 + 2 + 2 + ... 2 = 100 misses
Total Misses for a Join	15000 misses	450 misses	300 misses
Total Query Time (Assume DAT = 10ms MAT = 0 ms)	150000 ms (= 2.5 min)	4500 ms (= 4.5 sec)	3000 ms (= 3 sec)

Topics

- Disks
  - Performance Metrics
- Database File Structure
- Tuple Organization in Files
- Buffer Manager
- Conclusion

Administrivia 10/29

- **Happy Halloween!**
- Some spooky reminders:
  - Project proposal due Monday 11/1
  - Homework 5 due Friday 11/5
- Last time...
  - Files => Blocks => Disk Sectors
    - Importance of blocks arranged contiguously on disk
    - Importance of file access pattern (sequential vs. irregular)
- Today
  - Modeling data access time
  - Disk scheduling

Administrivia 11/1

- Reminders:
  - Project proposals due tonight!
  - Hwk 5 due Friday
- Last time...
  - Modeling disk access time
- Today
  - Disk scheduling policies
  - Buffer/cache management