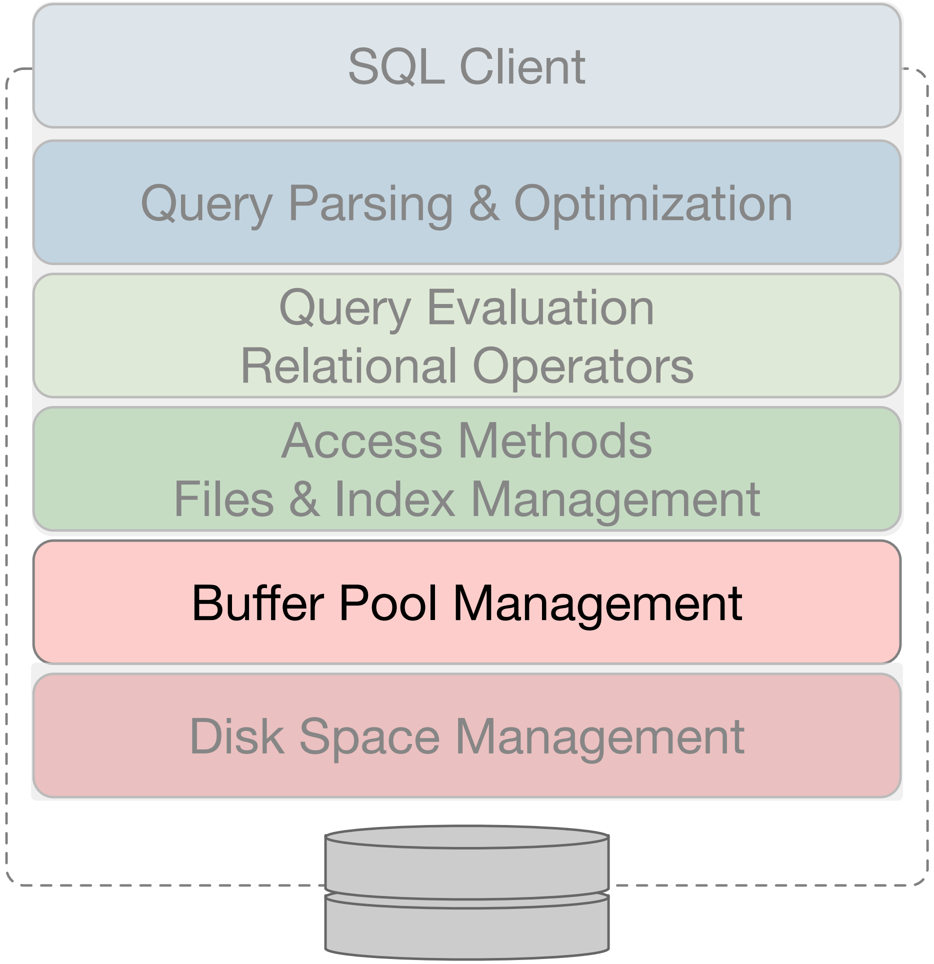
The Buffer Manager

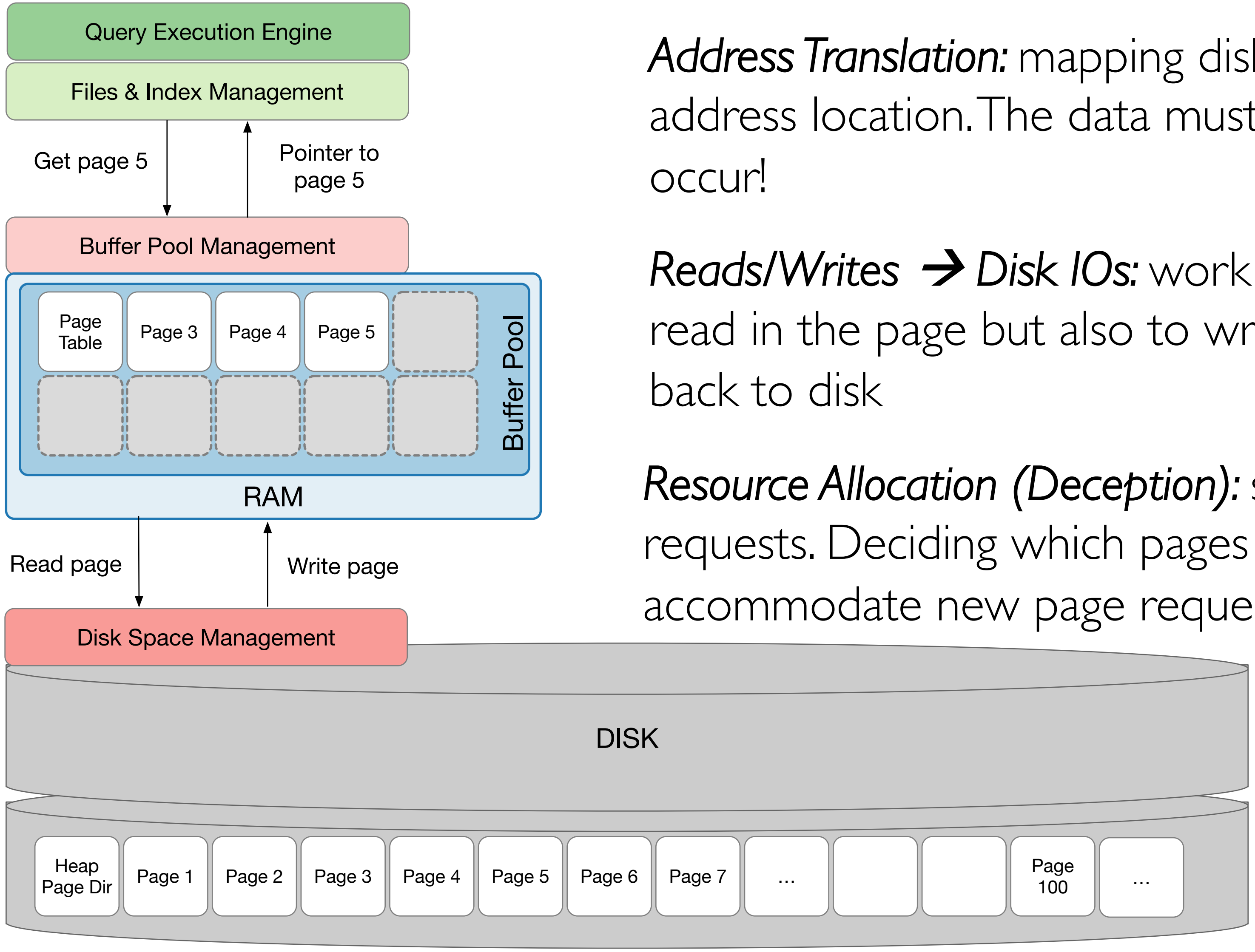


The Buffer Manager

Provides the *illusion* of accessing and modifying disk pages in memory.

Transforms page requests (reads, writes) from upper levels into storage layer IO requests.

What does this illusion entail?



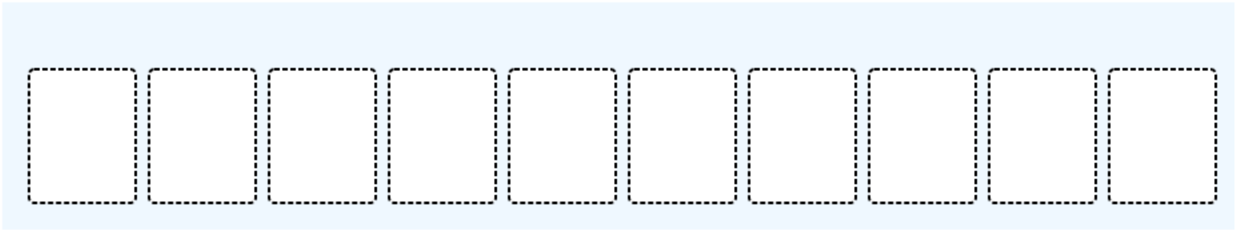
Address Translation: mapping disk page requests to a memory address location. The data must be in RAM for any processing to occur!

Reads/Writes → Disk IOs: work with the disk space manager to read in the page but also to write out modified "dirty" pages back to disk

Resource Allocation (Deception): small buffer, large disk, many page requests. Deciding which pages to remove from memory to accommodate new page requests.

The Buffer Pool & The Page Table

The Buffer Pool



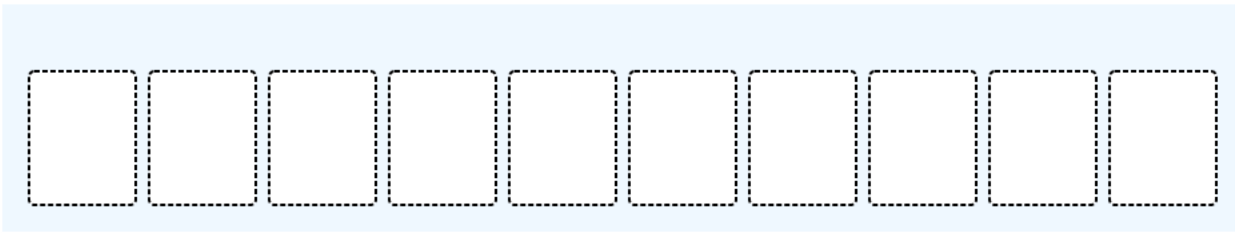
Buffer pool: Large range of memory, malloced at DBMS server boot time. Divided into frames. Each frame can hold a page from disk.

FrameID	PageID	Dirty?	Pin Count
0			
1			
2			
3			
4			
5			
6			
7			
8			
9			

Page table: Small array malloced at DBMS server boot time to keep track of the buffer state.

Hash table index on PageID: to get the frame in which the page is loaded.

The Buffer Pool

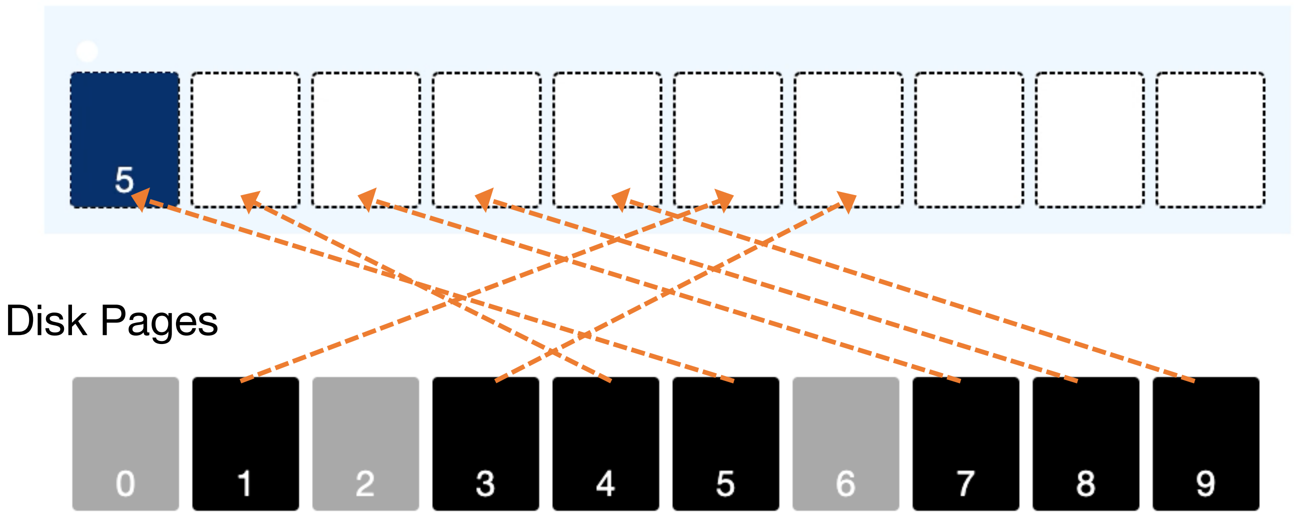


Access Reference Stream

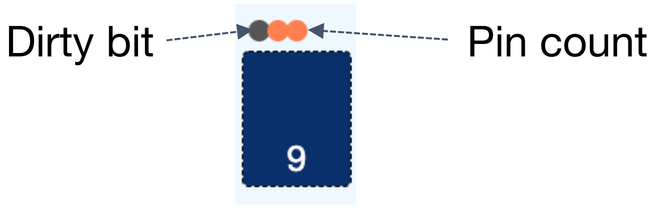
R 5 | R 4 | W 7 | R 8 | R 9 | W 1 | W 9 | R 8 | R 4 | R 3 | Replay ↻

Read & write requests arrive from different transactions. Each requestor pins while in use and unpins pages immediately after.

The Buffer Pool

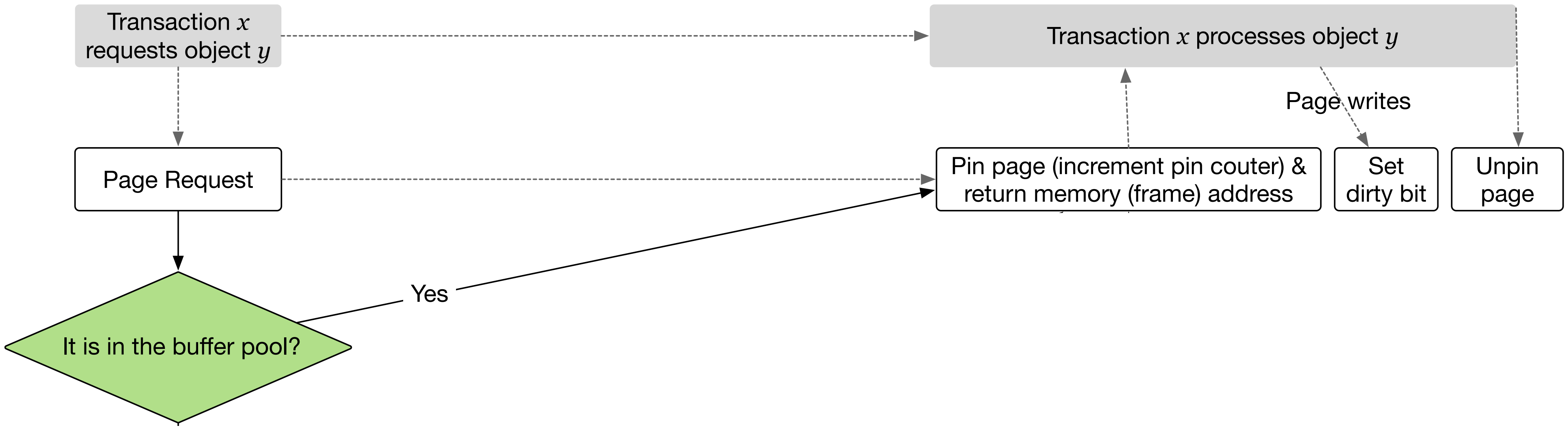


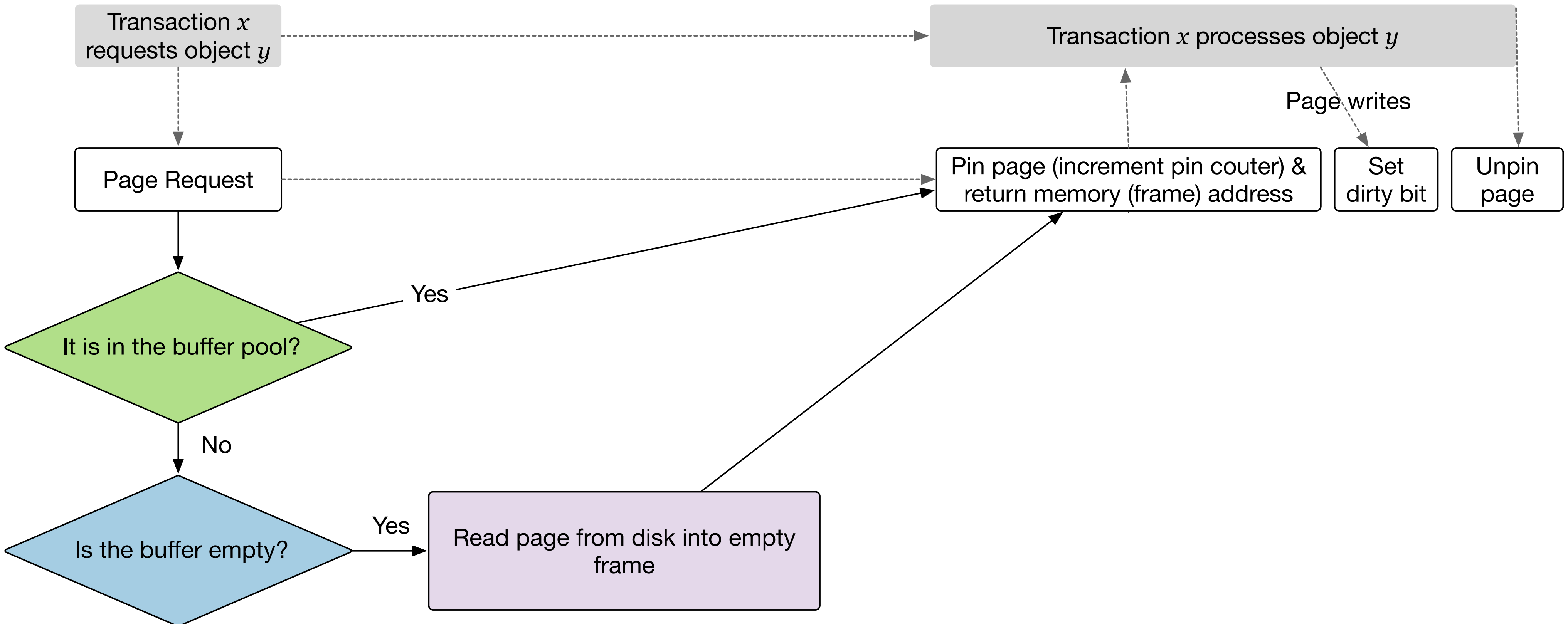
FrameID	PageID	Dirty?	Pin Count
0	5	0	0
1	4	0	0
2	7	1	1
3	8	0	1
4	9	1	2
5	1	1	0
6	3	0	0
7			
8			
9			

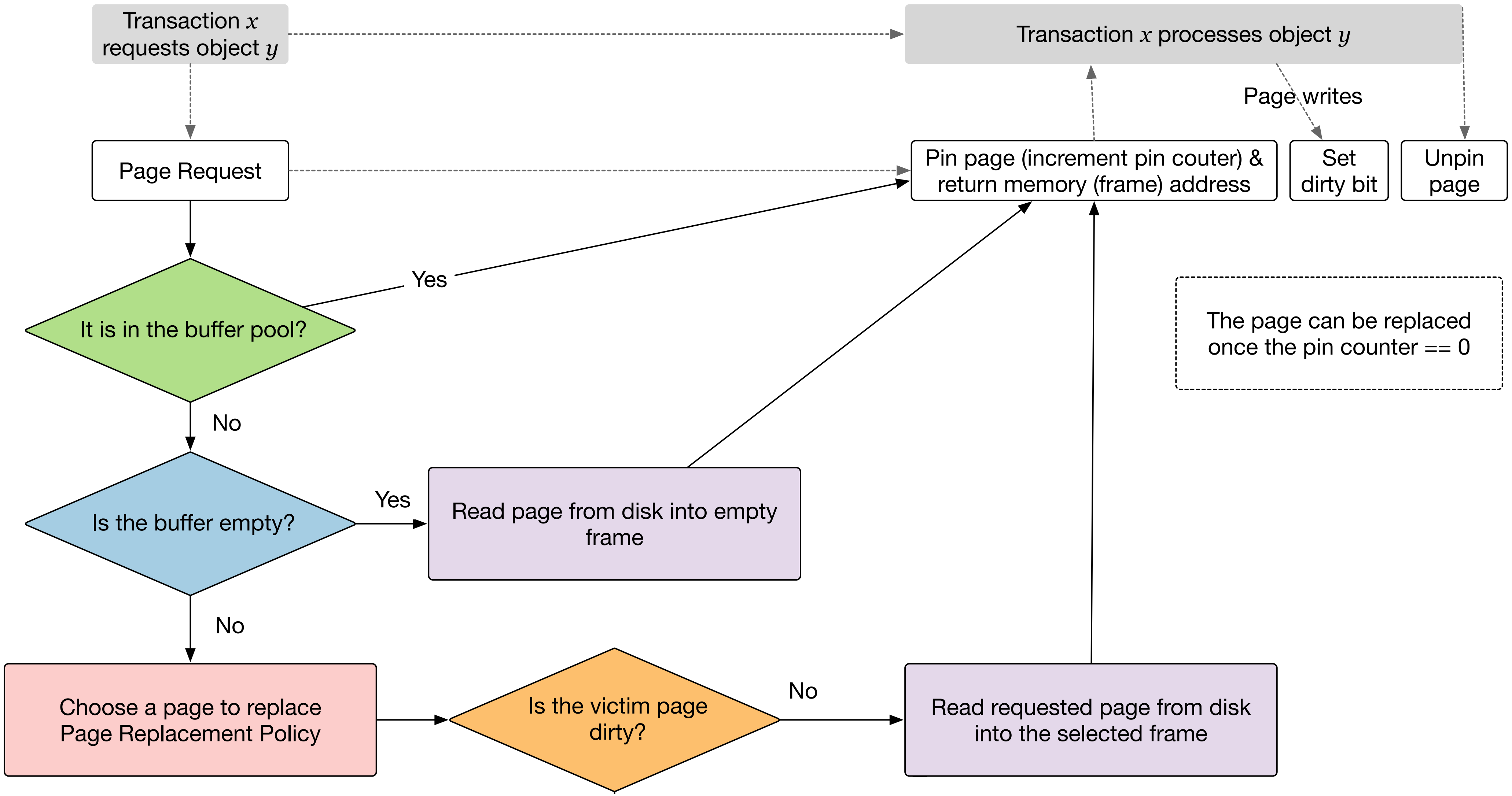


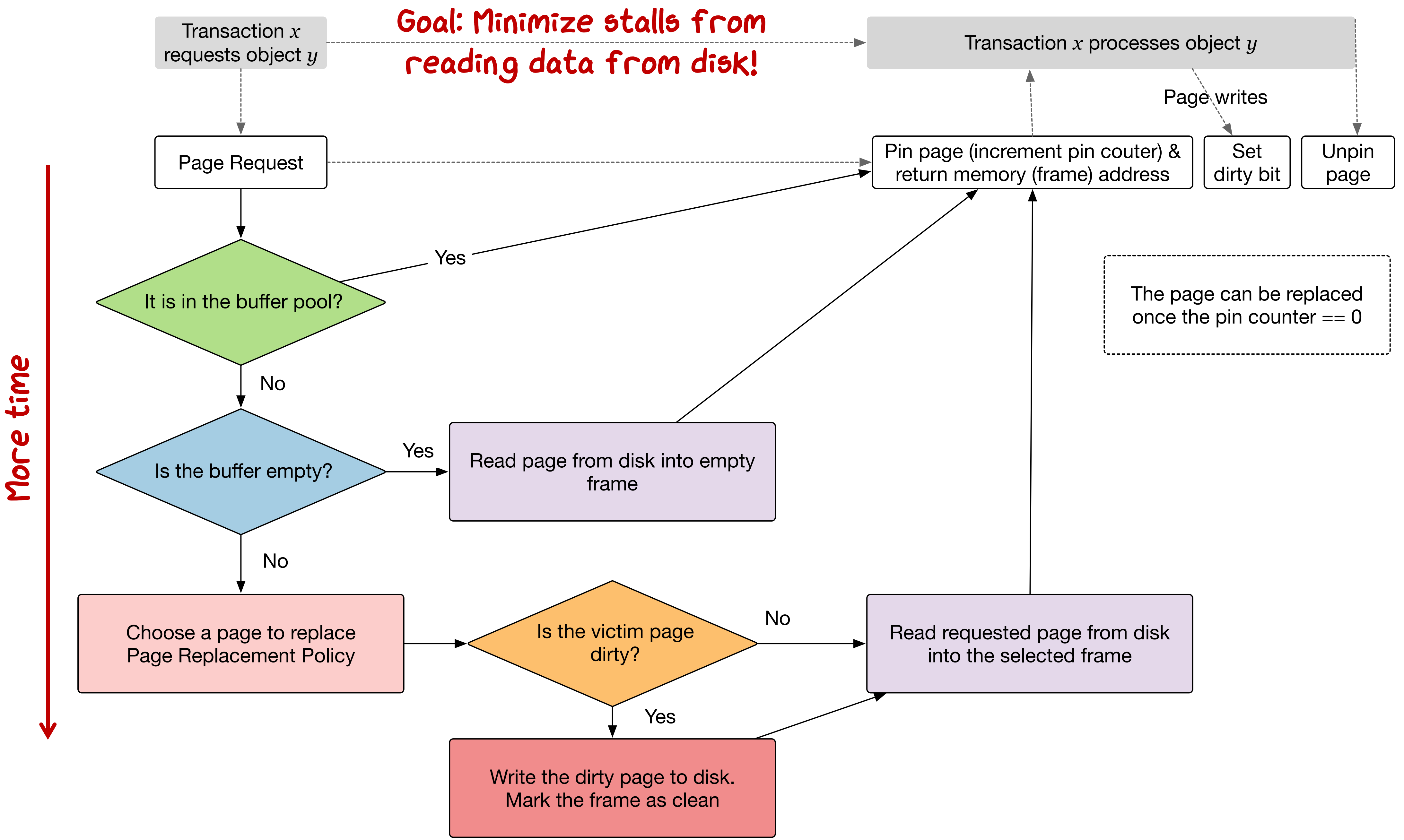
Why do we use a dirty bit?

Why do we store a pin count?


Transaction x requests object y
Transaction x processes object y
Page writes
Page Request
Pin page (increment pin couter) & return memory (frame) address
Set dirty bit
Unpin page
Yes
It is in the buffer pool?


Transaction x requests object y
Transaction x processes object y
Page writes
Page Request
Pin page (increment pin couter) & return memory (frame) address
Set dirty bit
Unpin page
Yes
It is in the buffer pool?
No
Is the buffer empty?
Yes
Read page from disk into empty frame


Transaction x requests object y
Transaction x processes object y
Page writes
Page Request
Pin page (increment pin couter) & return memory (frame) address
Set dirty bit
Unpin page
Yes
It is in the buffer pool?
The page can be replaced once the pin counter == 0
No
Yes
Is the buffer empty?
Read page from disk into empty frame
No
Choose a page to replace Page Replacement Policy
Is the victim page dirty?
No
Read requested page from disk into the selected frame


Goal: Minimize stalls from reading data from disk!
Transaction x requests object y
Transaction x processes object y
Page writes
Page Request
Pin page (increment pin couter) & return memory (frame) address
Set dirty bit
Unpin page
It is in the buffer pool?
Yes
No
The page can be replaced once the pin counter == 0
Is the buffer empty?
Yes
Read page from disk into empty frame
No
More time
Choose a page to replace Page Replacement Policy
Is the victim page dirty?
No
Read requested page from disk into the selected frame
Yes
Write the dirty page to disk. Mark the frame as clean

Page Replacement Policies

Design Goals of a Page Replacement Policy

Minimizes Cache Misses

If you knew the future, you could optimize the choice of pages to keep/replace to minimize your disk IOs

Minimal Overhead

The implementation of the policy needs to be fast: cannot afford to wait for an expensive algorithm to run.

You need to minimize the meta-data you store for each page as well.

LRU & CLOCK

Least Recently Used (LRU)

Maintain a timestamp of when each page was last accessed.

When the DBMS needs to evict a page, select the one with the oldest timestamp.

Minimizes Cache Misses

Intuition: if we haven't accessed a page in while, we probably won't access it again. Why? Temporal locality

Minimal Overhead

Use a min-heap data structure to efficiently search for the "least recently used" page to replace.

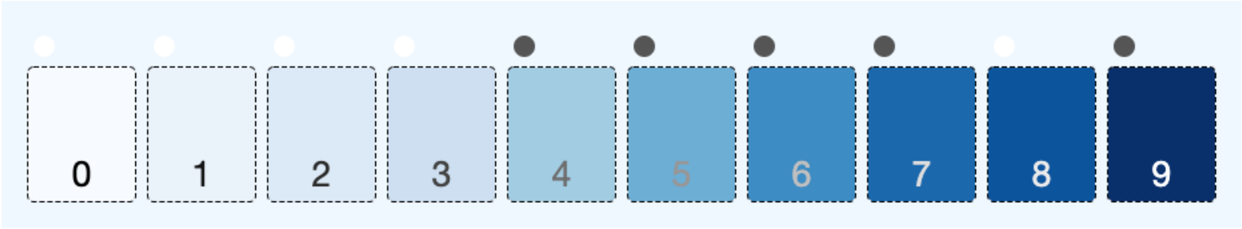
Access Reference Stream

R 0 R 1 R 2 R 3 W 4 W 5 W 6 W 7 R 8 W 9
W 10 R 11 R 12 R 13 R 14 R 5 W 6 R 7 W 8 W 9
R 15 R 16 R 17 R 18 R 19 R 5 R 6 W 7 R 8 R 9
Replay ↻

Buffer/Disk Statistics

Cache Hits	0
Cache Misses	10
Disk Writes	0

The Buffer Pool



FrameID	PageID	Dirty?	Pin Count	Last Accessed
0	0	0	0	16
1	1	0	0	19
2	2	0	0	23
3	3	0	0	27
4	4	1	0	28
5	5	1	0	32
6	6	1	0	36
7	7	1	0	37
8	8	0	0	39
9	9	1	0	40

LRU in action

Access Reference Stream

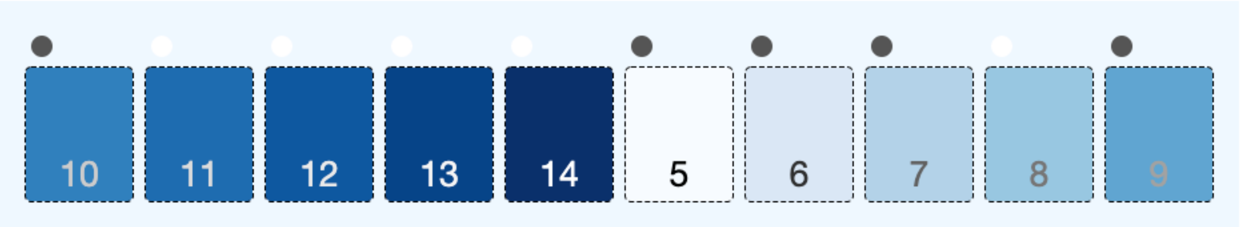
R 0 R 1 R 2 R 3 W 4 W 5 W 6 W 7 R 8 W 9
W 10 R 11 R 12 R 13 R 14 R 5 W 6 R 7 W 8 W 9
R 15 R 16 R 17 R 18 R 19 R 5 R 6 W 7 R 8 R 9
Replay ↻

Buffer/Disk Statistics

Cache Hits	0
Cache Misses	15
Disk Writes	1

FrameID	PageID	Dirty?	Pin Count	Last Accessed
0	10	1	0	45
1	11	0	0	53
2	12	0	0	57
3	13	0	0	59
4	14	0	0	62
5	5	1	0	32
6	6	1	0	36
7	7	1	0	37
8	8	0	0	39
9	9	1	0	40

The Buffer Pool



LRU in action

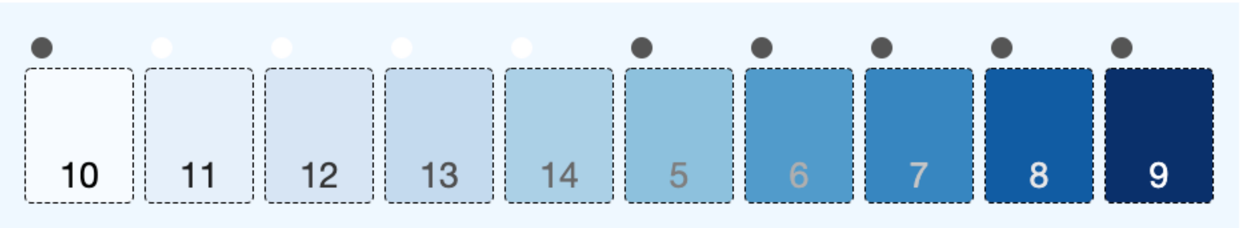
Access Reference Stream

R 0 R 1 R 2 R 3 W 4 W 5 W 6 W 7 R 8 W 9
W 10 R 11 R 12 R 13 R 14 R 5 W 6 R 7 W 8 **W 9**
R 15 R 16 R 17 R 18 R 19 R 5 R 6 W 7 R 8 R 9
Replay ↻

Buffer/Disk Statistics

Cache Hits	5
Cache Misses	15
Disk Writes	1

The Buffer Pool



FrameID	PageID	Dirty?	Pin Count	Last Accessed
0	10	1	0	45
1	11	0	0	53
2	12	0	0	57
3	13	0	0	59
4	14	0	0	62
5	5	1	0	68
6	6	1	0	69
7	7	1	0	71
8	8	0	0	73
9	9	1	0	76

LRU in action

Access Reference Stream

R 0	R 1	R 2	R 3	W 4	W 5	W 6	W 7	R 8	W 9
W 10	R 11	R 12	R 13	R 14	R 5	W 6	R 7	W 8	W 9
R 15	R 16	R 17	R 18	R 19	R 5	R 6	W 7	R 8	R 9

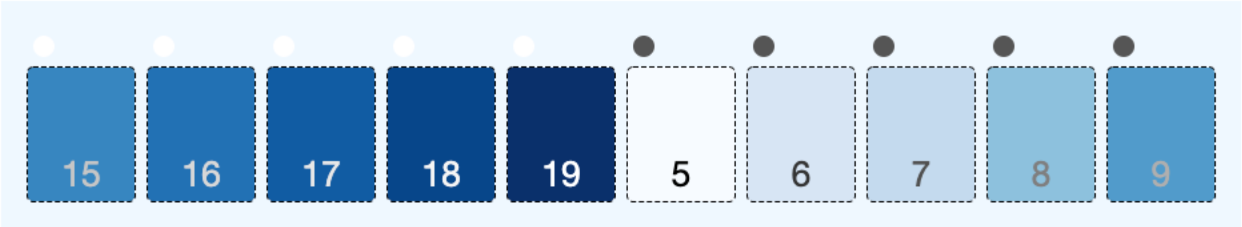
Replay ↻

Buffer/Disk Statistics

Cache Hits	5
Cache Misses	20
Disk Writes	2

FrameID	PageID	Dirty?	Pin Count	Last Accessed
0	15	1	0	81
1	16	0	0	84
2	17	0	0	88
3	18	0	0	93
4	19	0	0	94
5	5	1	0	68
6	6	1	0	69
7	7	1	0	71
8	8	0	0	73
9	9	1	0	76

The Buffer Pool



LRU in action

Access Reference Stream

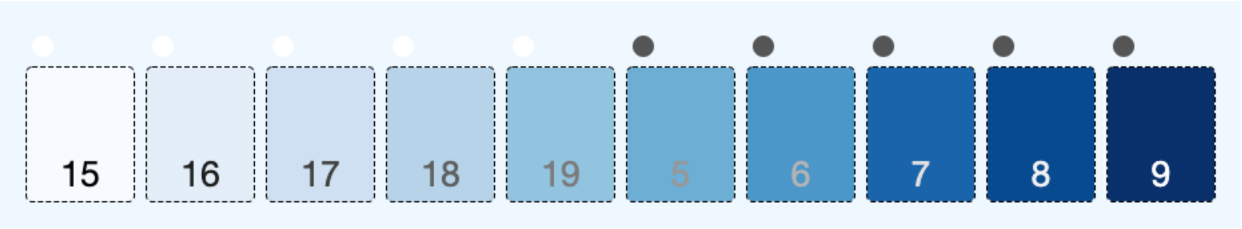
R 0 | R 1 | R 2 | R 3 | W 4 | W 5 | W 6 | W 7 | R 8 | W 9
W 10 | R 11 | R 12 | R 13 | R 14 | R 5 | W 6 | R 7 | W 8 | W 9
R 15 | R 16 | R 17 | R 18 | R 19 | R 5 | R 6 | W 7 | R 8 | **R 9**

Replay ↻

Buffer/Disk Statistics

Cache Hits 10
Cache Misses 20
Disk Writes 2

The Buffer Pool



FrameID	PageID	Dirty?	Pin Count	Last Accessed
0	10	1	0	81
1	11	0	0	84
2	12	0	0	88
3	13	0	0	93
4	14	0	0	94
5	5	1	0	100
6	6	1	0	102
7	7	1	0	106
8	8	0	0	109
9	9	1	0	111

LRU in action

LRU → CLOCK

Approximate LRU

What if instead of using 4-8 bytes to store time, we used a single bit

LRU --- Minimal Overhead?

Do not store a separate timestamp per page!

Just store a reference bit.

Scan through pages in the page table as if they were in a circular buffer with a "clock hand":

- If ref bit is 1, set to 0
- If ref bit is 0, evict!

Access Reference Stream

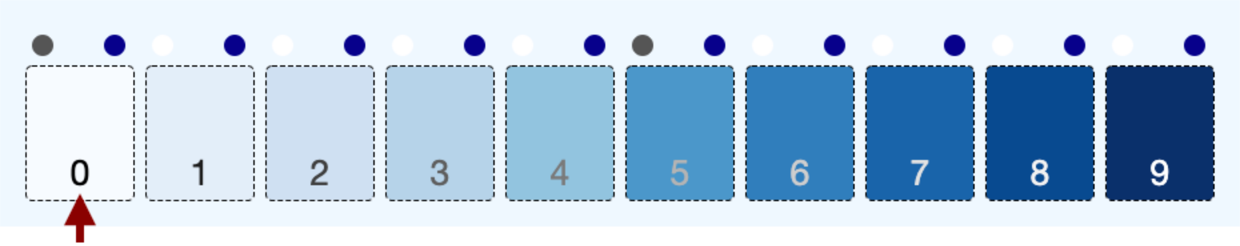
W 0	R 1	R 2	R 3	R 4	W 5	R 6	R 7	R 8	R 9
W 10	W 11	R 12	R 13	R 14	W 5	W 6	R 7	W 8	R 9
R 15	R 16	R 17	W 18	R 19	R 5	W 6	R 7	R 8	R 9

Replay ↻

Buffer/Disk Statistics

Cache Hits	0
Cache Misses	10
Disk Writes	0

The Buffer Pool



FrameID	PageID	Dirty?	Pin Count	Ref Bit
0	0	1	0	1
1	1	0	0	1
2	2	0	0	1
3	3	0	0	1
4	4	0	0	1
5	5	1	0	1
6	6	1	0	1
7	7	1	0	1
8	8	0	0	1
9	9	1	0	1

CLOCK in action

Access Reference Stream

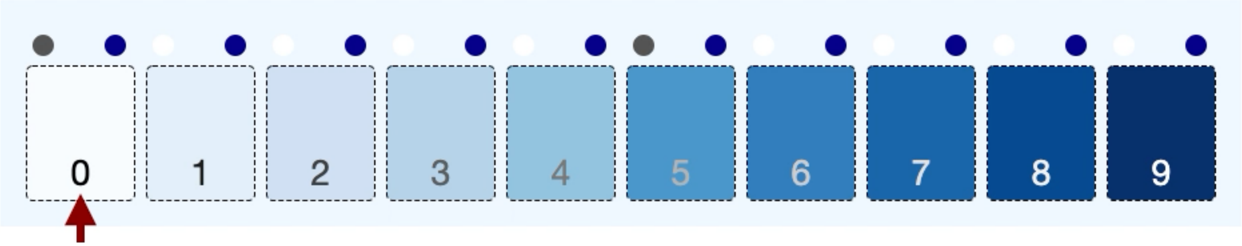
W 0 | R 1 | R 2 | R 3 | R 4 | W 5 | R 6 | R 7 | R 8 | R 9
W 10 | W 11 | R 12 | R 13 | R 14 | W 5 | W 6 | R 7 | W 8 | R 9
R 15 | R 16 | R 17 | W 18 | R 19 | R 5 | W 6 | R 7 | R 8 | R 9

Replay ↻

Buffer/Disk Statistics

Cache Hits	0
Cache Misses	10
Disk Writes	0

The Buffer Pool



FrameID	PageID	Dirty?	Pin Count	Ref Bit
0	10	1	0	1
1	1	0	0	0
2	2	0	0	0
3	3	0	0	0
4	4	0	0	0
5	5	1	0	0
6	6	0	0	0
7	7	0	0	0
8	8	0	0	0
9	9	0	0	0

CLOCK in action

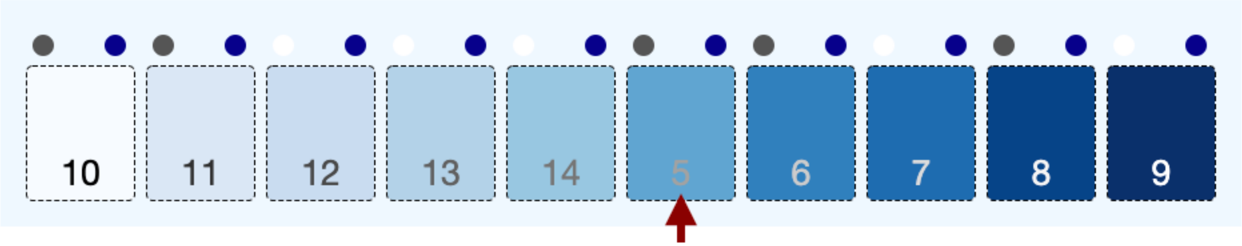
Access Reference Stream

W 0 R 1 R 2 R 3 R 4 W 5 R 6 R 7 R 8 R 9
W 10 W 11 R 12 R 13 R 14 W 5 W 6 R 7 W 8 **R 9**
R 15 R 16 R 17 W 18 R 19 R 5 W 6 R 7 R 8 R 9
Replay ↻

Buffer/Disk Statistics

Cache Hits	5
Cache Misses	15
Disk Writes	1

The Buffer Pool



FrameID	PageID	Dirty?	Pin Count	Ref Bit
0	10	1	0	1
1	11	1	0	1
2	12	0	0	1
3	13	0	0	1
4	14	0	0	1
→ 5	5	1	0	1
6	6	1	0	1
7	7	0	0	1
8	8	1	0	1
9	9	0	0	1

CLOCK in action

Access Reference Stream

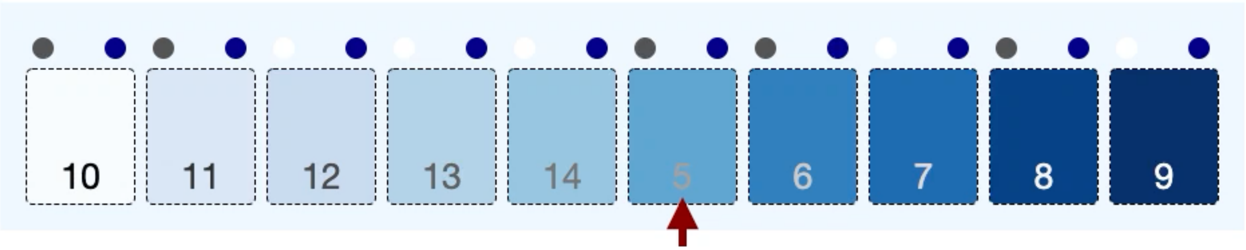
W 0 | R 1 | R 2 | R 3 | R 4 | W 5 | R 6 | R 7 | R 8 | R 9
W 10 | W 11 | R 12 | R 13 | R 14 | W 5 | W 6 | R 7 | W 8 | R 9
R 15 | R 16 | R 17 | W 18 | R 19 | R 5 | W 6 | R 7 | R 8 | R 9

Replay ↻

Buffer/Disk Statistics

Cache Hits 5
Cache Misses 15
Disk Writes 1

The Buffer Pool



FrameID	PageID	Dirty?	Pin Count	Ref Bit
0	10	1	0	0
1	11	1	0	0
2	12	0	0	0
3	13	0	0	0
4	14	0	0	0
5	15	0	0	1
6	6	1	0	0
7	7	0	0	0
8	8	1	0	0
9	9	0	0	0

CLOCK in action

Access Reference Stream

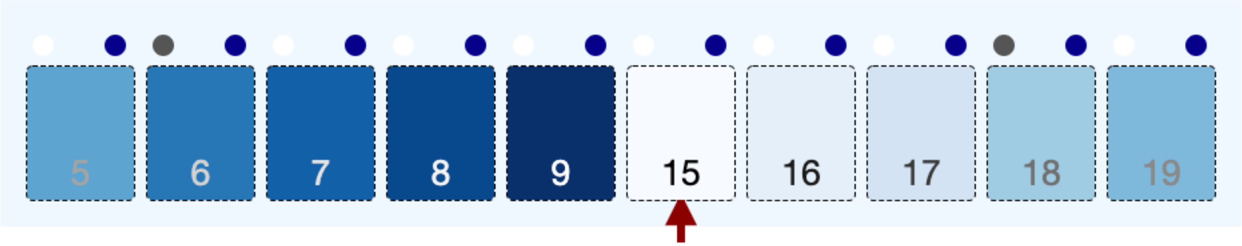
W 0 | R 1 | R 2 | R 3 | R 4 | W 5 | R 6 | R 7 | R 8 | R 9
W 10 | W 11 | R 12 | R 13 | R 14 | W 5 | W 6 | R 7 | W 8 | R 9
R 15 | R 16 | R 17 | W 18 | R 19 | R 5 | W 6 | R 7 | R 8 | R 9

Replay ↻

Buffer/Disk Statistics

Cache Hits	5
Cache Misses	25
Disk Writes	6

The Buffer Pool



FrameID	PageID	Dirty?	Pin Count	Ref Bit
0	5	1	0	1
1	6	1	0	1
2	7	0	0	1
3	8	0	0	1
4	9	0	0	1
→ 5	15	1	0	1
6	16	1	0	1
7	17	0	0	1
8	18	1	0	1
9	19	0	0	1

CLOCK in action

LRU or CLOCK

Minimizes Cache Misses

Intuition: if we haven't accessed a page in while, we probably won't access it again.

Minimizes cache misses for some workloads

Sequential Flooding

- A query sequentially scans a large file of many pages.
- The buffer is polluted with pages that are read once and then never again …

```
select * from R where id = 12;

select sum(a) from LargeTable;

select * from R where id = 18;
```

- or will be read again but soon after eviction in the case of repeated scans

```
select * from R, Y where R.id = Y.id;
```

- Certain join implementations (e.g. nested loop join) repeatedly scan the inner table.

MRU

Most Recently Used (LRU)

Maintain pages in order of last access.

When the DBMS needs to evict a page, select the one with the most recent timestamp!

Minimizes Cache Misses for Repeated Scans

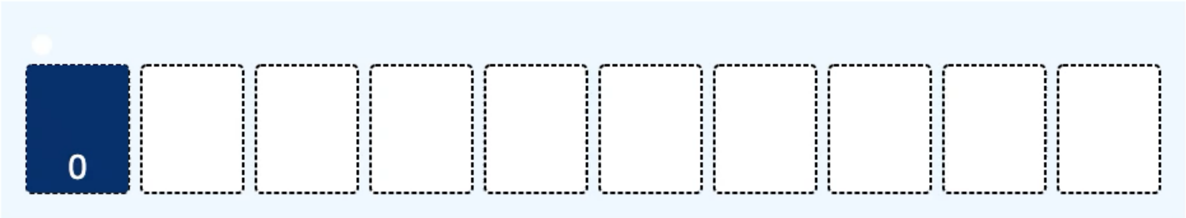
Intuition: The pages that we just read in are least likely to be read again (at least for a while!)

Access Reference Stream

R 0 R 1 R 2 R 3 R 4 R 5 R 6 R 7 R 8 R 9 R 10 R 11 R 12 R 13 R 14
R 0 R 1 R 2 R 3 R 4 R 5 R 6 R 7 R 8 R 9 R 10 R 11 R 12 R 13 R 14
Replay ↻

Repeated Scans

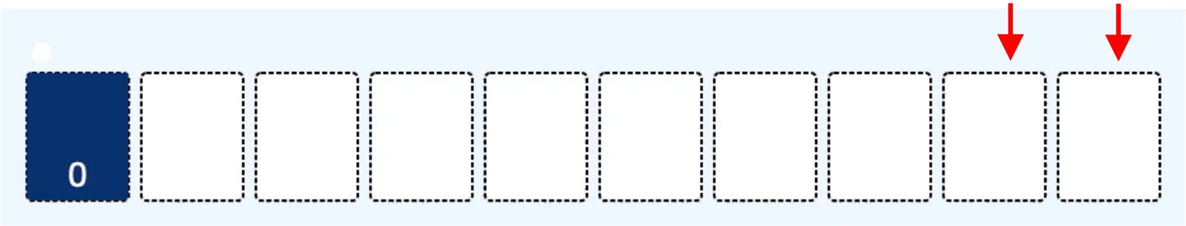
The Buffer Pool



Buffer/Disk Statistics

Cache Hits	0
Cache Misses	1
Disk Writes	0

The Buffer Pool



Buffer/Disk Statistics

Cache Hits	0
Cache Misses	1
Disk Writes	0

MRU vs. LRU

Policy Optimizations

Multiple Buffer Pools & Hints

Multiple buffer pools:

- Per type (index vs. database table file pools)
- Per transaction/query/database

Different policies per pool.

DBMS can provide hints on which pages to keep or to evict.

Top levels or root of an index are important, but leaf nodes may not be frequently accessed

Prefetching

For a given query, we can predict the access patten. So prefetch the pages into the buffer even before they are requested.

For example:

```
select sum(a)
from R;
```

is a sequential scan so prefetch a few pages at a time.

Likelihood of Re-access LRU-K (Often LRU-2)

DOI:10.1145/1538788.1538805

Article development led by acmqueue
queue.acm.org

Revisiting Gray and Putzolu's famous rule in the age of Flash.

BY GOETZ GRAEFE

The Five-Minute Rule 20 Years Later (and How Flash Memory Changes the Rules)

IN 1987, JIM Gray and Gianfranco Putzolu published their now-famous five-minute rule[15] for trading off memory and I/O capacity. Their calculation compares the cost of holding a record (or page) permanently in memory with the cost of performing disk I/O each time the record (or page) is accessed, using appropriate fractional prices of RAM chips and disk drives. The name of their rule refers to the break-even interval between accesses. If a record (or page) is accessed more often, it should be kept in memory; otherwise, it should remain on disk and be read when needed.

Based on then-current prices and performance characteristics of Tandem equipment, Gray and Putzolu found the price of RAM to hold a 1KB record was about equal to the (fractional) price of a disk drive required to access such a record every 400 seconds, which they rounded to five minutes. The break-even interval is about inversely proportional to the record size. Gray and Putzolu reported one hour for 100-byte records and two minutes for 4KB pages.

The five-minute rule was reviewed and renewed 10 years later.[14] Lots of prices and performance parameters had changed (for example, the price of RAM had tumbled from $5,000 to $15 per megabyte). Nonetheless, the break-even interval for 4KB pages was still around five minutes. The first goal of this article is to review the five-minute rule after another 10 years.

Of course, both previous articles acknowledged that prices and performance vary among technologies and devices at any point in time (RAM for mainframes versus mini-computers, SCSI versus IDE disks, and so on). Interested readers are invited to reevaluate the appropriate formulas for their environments and equipment. The values used here (in Table 1) are meant to be typical for 2007 technologies rather than universally accurate.

In addition to quantitative changes in prices and performance, qualitative changes already under way will affect the software and hardware architectures of servers and, in particular, database systems. Database software will change radically with the advent of new technologies: virtualization with hardware and software support, as well as higher utilization goals for physical machines; many-core processors and transactional memory supported both in programming environments and hardware;[20] deployment in containers housing thousands of processors and many terabytes of data;[17] and flash memory that fills the gap between traditional RAM and traditional rotating disks.

Flash memory falls between traditional RAM and persistent mass storage based on rotating disks in terms of acquisition cost, access

48 COMMUNICATIONS OF THE ACM | JULY 2009 | VOL. 52 | NO. 7

Keep it dirty / Background writes

When choosing which page to evict, you can *skip "dirty" pages* --- the lazy approach saves you a write during a page request.

A *background thread can flush the dirty pages* every now & then to disk giving you more clean pages on page requests that don't require "write".

We will examine writes again when looking at recovery and concurrency control.

Scan Sharing

Query A is scanning relation R and has scanned and evicted 5 pages already

Query B arrives and, also wants to scan relation R.

Query B jumps in with A and starts at A's current cursor sharing the scan.

Query B then requests the first 5 pages that it missed.

OS & DBMS Buffer Interactions

Most disk IO requests go through the OS. The OS maintains its own filesystem cache.

Why can't we just rely on the FS cache?

Portability: Different FS cache use different page replacement policies. This leads to different performance on different OS and a DBMS needs better control.

Force writes: We will see later that recovery protocols require the DBMS to enforce page flushes to disk and the OS may delay such requests.

Prefetching: DBMS has more information on the access patterns of different queries and benefits from prefetching and page hints. The OS FS is general purpose.