

# COMP 421: Files & Databases

## Lecture 5: Buffer Pool Manager

# Last Class

**Problem #1:** How the DBMS represents the database in files on disk.

**Problem #2:** How the DBMS manages its memory and move data back-and-forth from disk.

# Database Storage

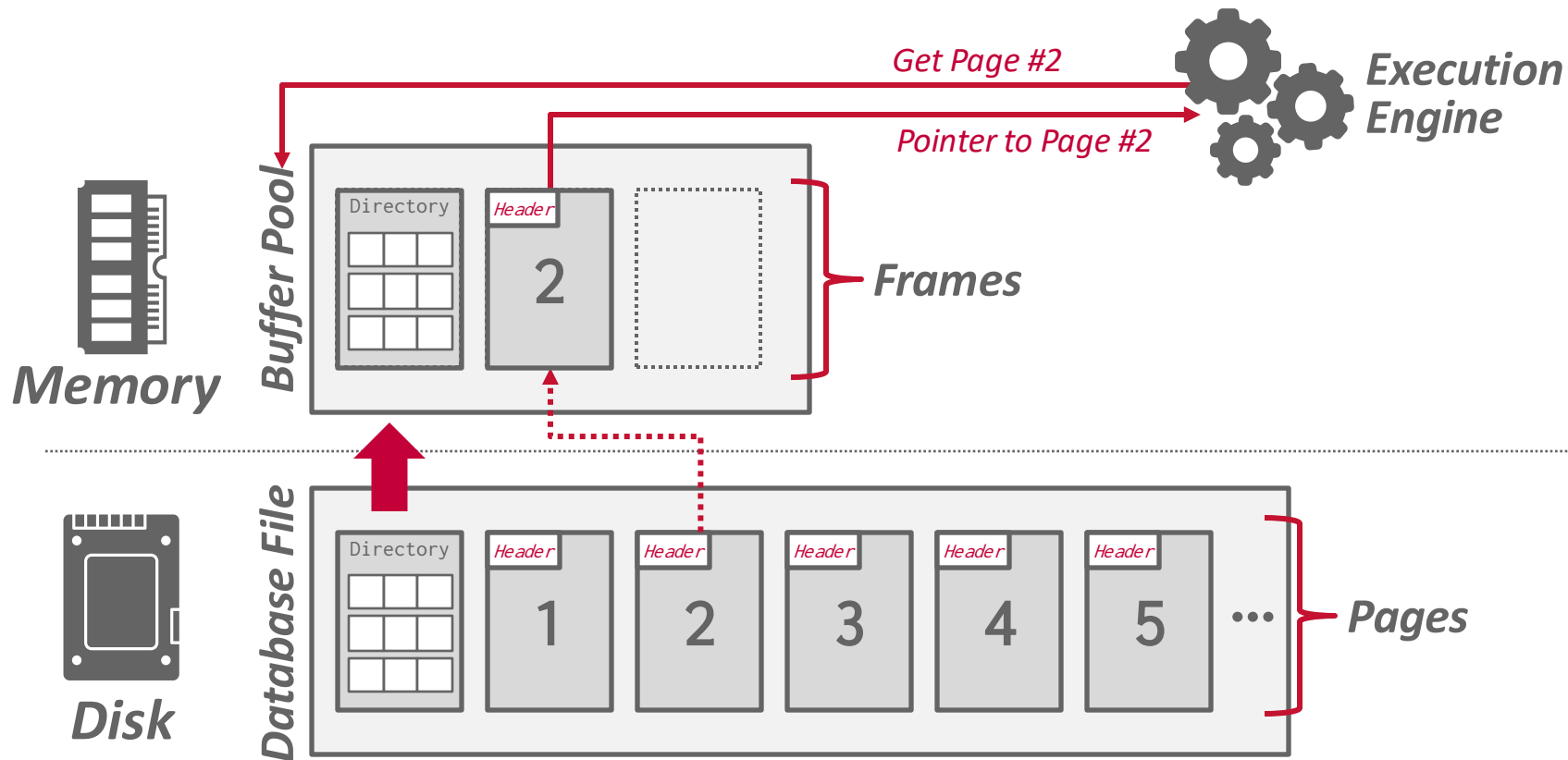
## **Spatial Control:**

- Where to write pages on disk.
- The goal is to keep pages that are used together often as physically close together as possible on disk.

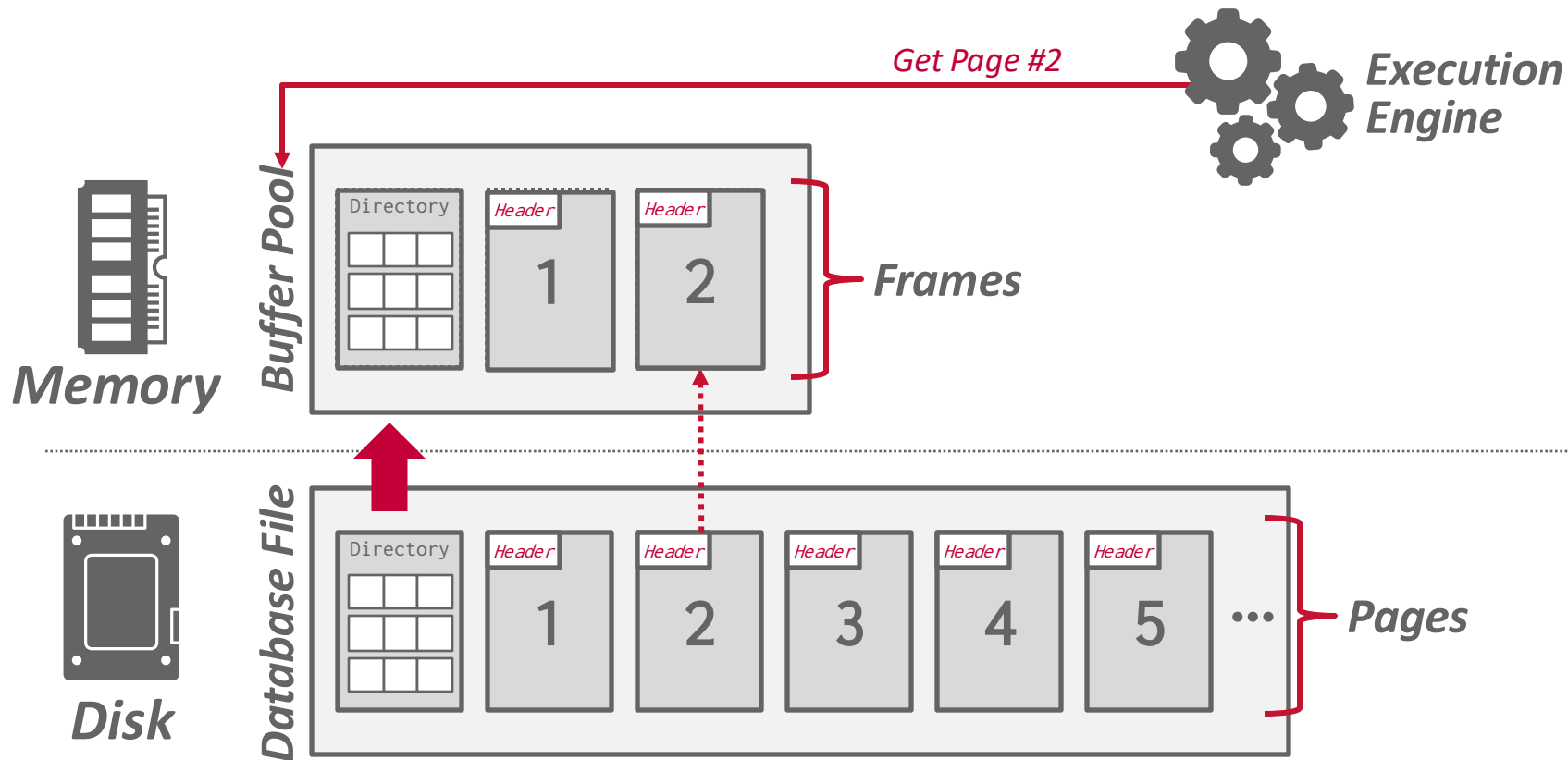
## **Temporal Control:**

- When to read pages into memory, and when to write them to disk.
- The goal is to minimize the number of stalls from having to read data from disk.

# Disk-oriented DBMS



# Disk-oriented DBMS



# Other Memory Pools

The DBMS needs memory for things other than just tuples and indexes.

These other memory pools may not always be backed by disk. Depends on implementation.

- Sorting + Join Buffers
- Query Caches
- Maintenance Buffers
- Log Buffers
- Dictionary Caches



# Today's Agenda

Buffer Pool Manager

Why **mmap** Will Murder Your DBMS

Disk I/O Scheduling

Replacement Policies

Optimizations

# Buffer Pool Organization

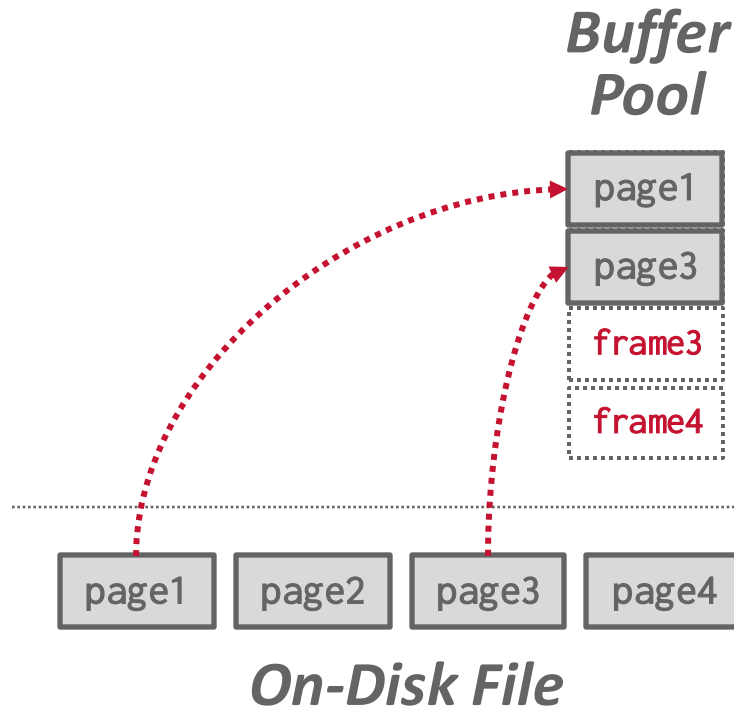
Memory region organized as an array of fixed-size pages.

An array entry is called a frame.

When the DBMS requests a page, an exact copy is placed into one of these frames.

When page is written in memory, it is marked "dirty"

- Dirty pages are buffered and not written to disk immediately
- Write-Back Cache





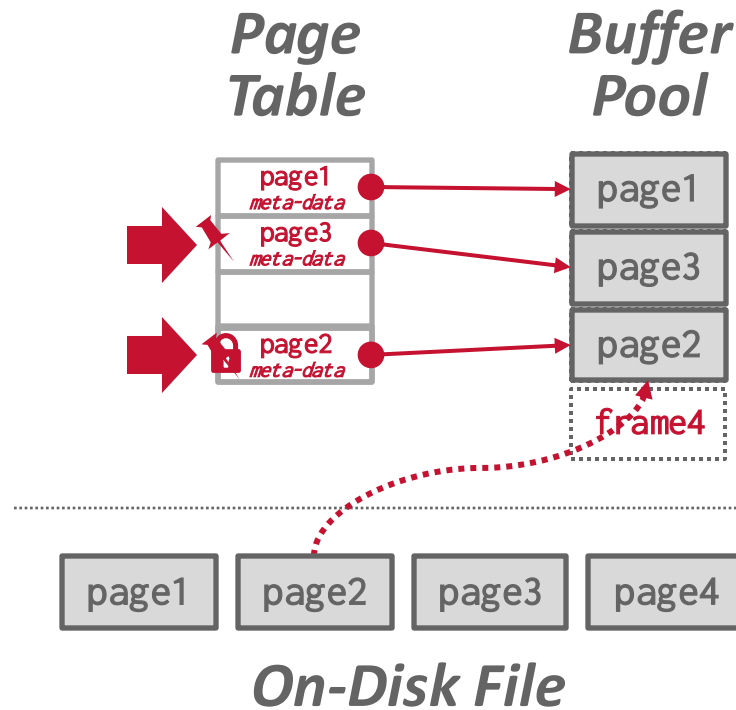
# Buffer Pool Metadata

The page table keeps track of pages that are currently in memory.

→ Usually a fixed-size hash table protected with latches to ensure thread-safe access.

Additional meta-data per page:

- **Dirty Flag**
- **Pin/Reference Counter**
- **Access Tracking Information**



# Locks vs. Latches

## Locks:

- Protects the database's logical contents from other transactions.
- Held for transaction duration.
- Need to be able to rollback changes.

## Latches:

- Protects the critical sections of the DBMS's internal data structure from other threads.
- Held for operation duration.
- Do not need to be able to rollback changes.

← **Mutex**

# Page Table vs. Page Directory

The **page directory** is the mapping from page ids to page locations in the database files.

→ All changes must be recorded on disk to allow the DBMS to find on restart.

The **page table** is the mapping from page ids to a copy of the page in buffer pool frames.

→ This is an in-memory data structure that does not need to be stored on disk.

# The Dark Side



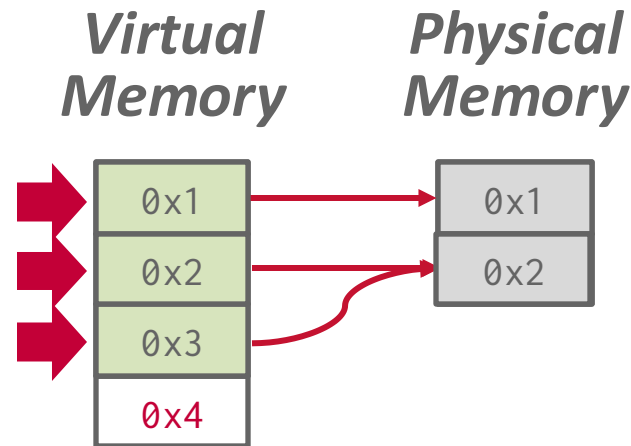
# Virtual Memory Crash Course

This is not a full intro! Take COMP 530!

Main purpose is to create indirection between "virtual" (logical) memory addresses and "physical" memory

- Ease of programming
- Process isolation/security

One of the great ideas in CS systems, beyond the scope of COMP 421



*Extra "swap" space on disk*

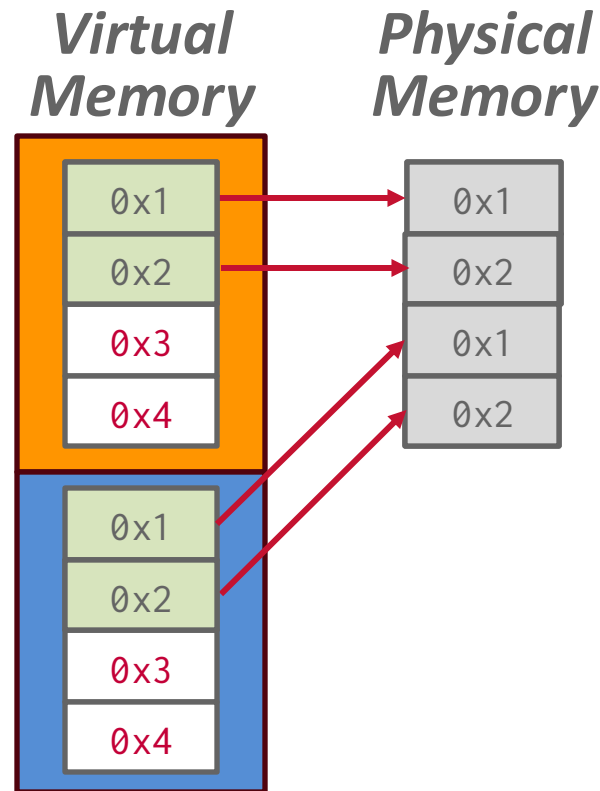
# Virtual Memory Crash Course

This is not a full intro! Take COMP 530!

Main purpose is to create indirection between "virtual" (logical) memory addresses and "physical" memory

- Ease of programming
- Process isolation/security

One of the great ideas in CS systems, beyond the scope of COMP 421

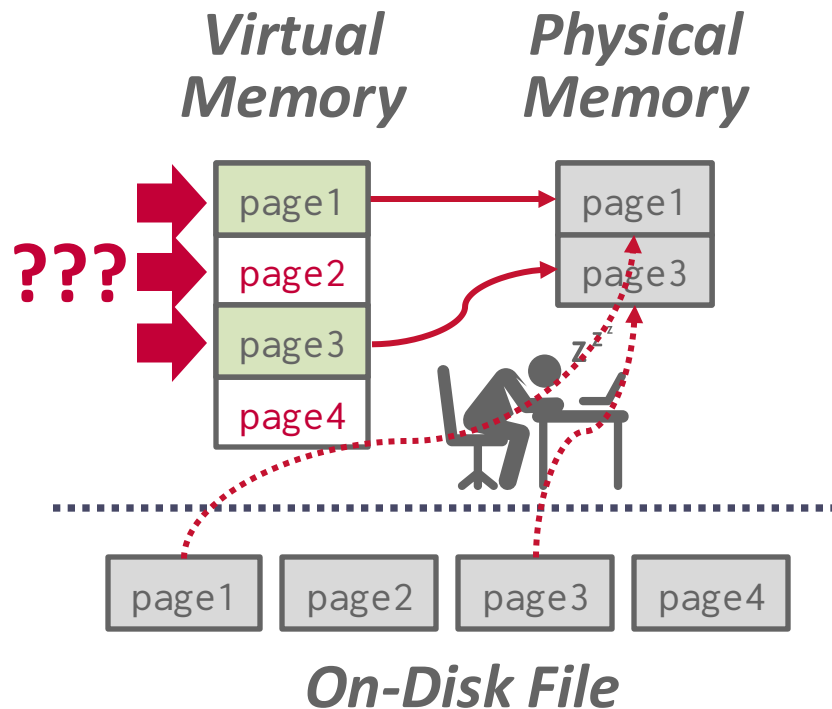


# Why Not Use The OS?

Use OS memory mapping (**mmap**) to store the contents of a file into the address space of a program.

OS is responsible for moving file pages in and out of memory, so the DBMS doesn't need to worry about it.

What if DBMS allows multiple threads to access **mmap** files to hide page fault stalls?



# Memory Mapped I/O Problems

## Problem #1: Transaction Safety

→ OS can flush dirty pages at any time.

## Problem #2: I/O Stalls

→ DBMS doesn't know which pages are in memory. The OS will stall a thread on page fault.

## Problem #3: Error Handling

→ Difficult to validate pages. Any access can cause a **SIGBUS** that the DBMS must handle.

## Problem #4: Performance Issues

→ OS data structure contention. (also, TLB shootdowns.)



# Why Not Use The OS?

There are some solutions to some of these problems:

- **madvise**: Tell the OS how you expect to read certain pages.
- **mlock**: Tell the OS that memory ranges cannot be paged out.
- **msync**: Tell the OS to flush memory ranges out to disk.

Using these syscalls to get the OS to behave correctly is just as onerous as managing memory yourself.

*Full Usage*



*Partial Usage*



# Why Not Use T

DBMS (almost) always wants to do itself and can do a better job than

- Flushing dirty pages to disk in the background
- Specialized prefetching.
- Buffer replacement policy.
- Thread/process scheduling.

The OS is not your friend.

## Are You Sure You Want to Use MMAP in Your Database Management System?

Andrew Crotty  
Carnegie Mellon University  
andrewcr@cs.cmu.edu

Viktor Leis  
University of Erlangen-Nuremberg  
viktor.leis@fau.de

Andrew Pavlo  
Carnegie Mellon University  
pavlo@cs.cmu.edu

### ABSTRACT

Memory-mapped (mmap) file I/O is an OS-provided feature that maps the contents of a file on secondary storage into a program's address space. The program then accesses pages via pointers as if the file resided entirely in memory. The OS transparently loads pages only when the program references them and automatically evicts pages if memory fills up.

mmap's perceived ease of use has seduced database management system (DBMS) developers for decades as a viable alternative to implementing a buffer pool. There are, however, severe correctness and performance issues with mmap that are not immediately apparent. Such problems make it difficult, if not impossible, to use mmap correctly and efficiently in a modern DBMS. In fact, several popular DBMSs initially used mmap to support larger-than-memory databases but soon encountered these hidden perils, forcing them to switch to managing file I/O themselves after significant engineering costs. In this way, mmap and DBMSs are like coffee and spicy food: an unfortunate combination that becomes obvious after the fact.

Since developers keep trying to use mmap in new DBMSs, we wrote this paper to provide a warning to others that mmap is not a suitable replacement for a traditional buffer pool. We discuss the main shortcomings of mmap in detail, and our experimental analysis demonstrates clear performance limitations. Based on these findings, we conclude with a prescription for when DBMS developers might consider using mmap for file I/O.

### 1 INTRODUCTION

An important feature of disk-based DBMSs is their ability to support databases that are larger than the available physical memory. This functionality allows a user to query a database as if it resided entirely in memory, even if it does not fit at all on one. DBMSs achieve this illusion by reading pages of data from secondary storage (e.g., HDD, SSD) into memory on demand. If there is not enough memory for a new page, the DBMS will evict an existing page that is no longer needed in order to make room.

Traditionally, DBMSs implement the movement of pages between secondary storage and memory in a buffer pool, which interacts with secondary storage using system calls like read and write. These file I/O mechanisms copy data to and from a buffer in user space, with the DBMS maintaining complete control over how and when it transfers pages.

Alternatively, the DBMS can relinquish the responsibility of data movement to the OS, which maintains its own file mapping and

page cache. The POSIX mmap system call maps a file on secondary storage into the virtual address space of the caller (i.e., the DBMS), and the OS will then load pages lazily when the DBMS accesses them. To the DBMS, the database appears to reside fully in memory, but the OS handles all necessary paging behind the scenes rather than the DBMS's buffer pool.

On the surface, mmap seems like an attractive implementation option for managing file I/O in a DBMS. The most notable benefits are ease of use and low engineering cost. The DBMS no longer needs to track which pages are in memory, nor does it need to track how often pages are accessed or which pages are dirty. Instead, the DBMS can simply access disk-resident data via pointers as if it were accessing data in memory while leaving all low-level page management to the OS. If the available memory fills up, then the OS will free space for new pages by transparently evicting (ideally unneeded) pages from the page cache.

From a performance perspective, mmap should also have much lower overhead than a traditional buffer pool. Specifically, mmap does not incur the cost of explicit system calls (i.e., read/write) and avoids redundant copying to a buffer in user space because the DBMS can access pages directly from the OS page cache.

Since the early 1980s, these supposed benefits have enticed DBMS developers to forgo implementing a buffer pool and instead rely on the OS to manage file I/O [36]. In fact, the developers of several well-known DBMSs (see Section 2.3) have gone down this path, with some even touting mmap as a key factor in achieving good performance [20].

Unfortunately, mmap has a hidden dark side with many sordid problems that make it undesirable for file I/O in a DBMS. As we describe in this paper, these problems involve both data safety and system performance concerns. We contend that the engineering steps required to overcome them negate the purported simplicity of working with mmap. For these reasons, we believe that mmap adds too much complexity with no commensurate performance benefit and strongly urge DBMS developers to avoid using mmap as a replacement for a traditional buffer pool.

The remainder of this paper is organized as follows. We begin with a short background on mmap (Section 2), followed by a discussion of its main problems (Section 3) and our experimental analysis (Section 4). We then discuss related work (Section 5) and conclude with a summary of our guidance for when you might consider using mmap in your DBMS (Section 6).

### 2 BACKGROUND

This section provides the relevant background on mmap. We begin with a high-level overview of memory-mapped file I/O and the POSIX mmap API. Then, we discuss real-world implementations of mmap-based systems.

<https://db.cs.cmu.edu/mmap-cidr2022>

This paper is published under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution, provided that you attribute the original work to the authors and CIDR 2022, 12th Annual Conference on Innovative Data Systems Research (CIDR '22), January 9-12, 2022, Chaminade, USA.



# Buffer Replacement Policies

When the DBMS needs to free up a frame to make room for a new page, it must decide which page to evict from the buffer pool.

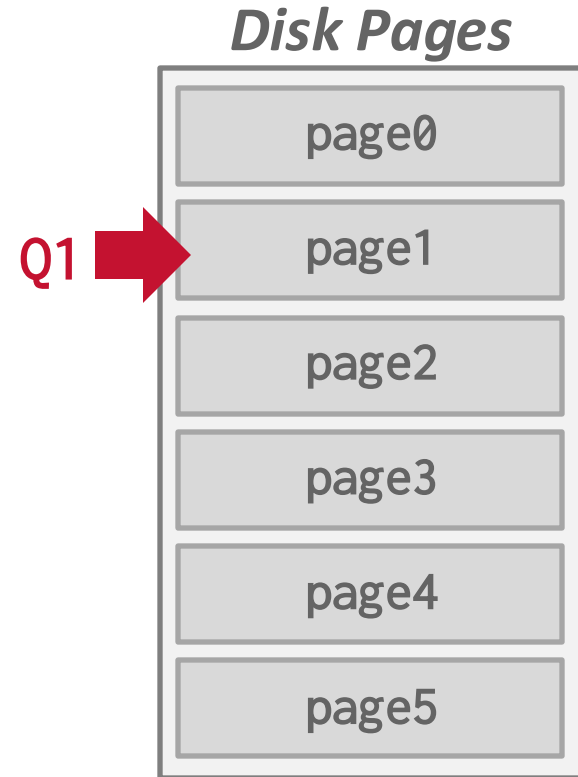
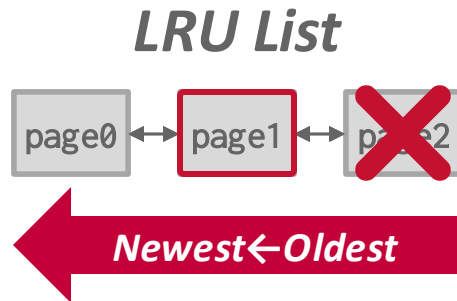
Goals:

- Correctness
- Accuracy
- Speed
- Meta-data overhead

# Least Recently Used (LRU)

Maintain a single timestamp of when each page was last accessed. When the DBMS needs to evict a page, select the one with the oldest timestamp.

→ Keep the pages in sorted order to reduce the search time on eviction.



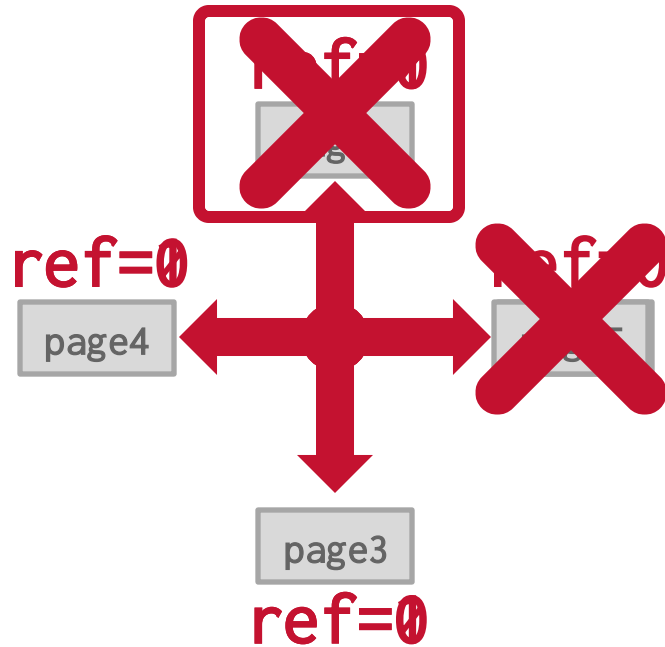
# CLOCK

Approximation of LRU that does not need a separate timestamp per page.

- Each page has a reference bit.
- When a page is accessed, set its bit to 1.

Organize pages in a circular buffer with a "clock hand" that sweeps over pages in order:

- As the hand visits each page, check if its bit is set to 1.
- If yes, set to zero. If no, then evict.



# Observation

LRU + CLOCK replacement policies are susceptible to **sequential flooding**.

- A query performs a sequential scan that reads every page in a table one or more times (e.g., blocked nested-loop joins).
- This pollutes the buffer pool with pages that are read once and then evicted

For scanning workloads, the ***most recently used*** page is often the best page to evict.

LRU + CLOCK only tracks when a page was last accessed, but not how often a page is accessed.

# Sequential Flooding

**Q1** SELECT \* FROM A WHERE id = 1

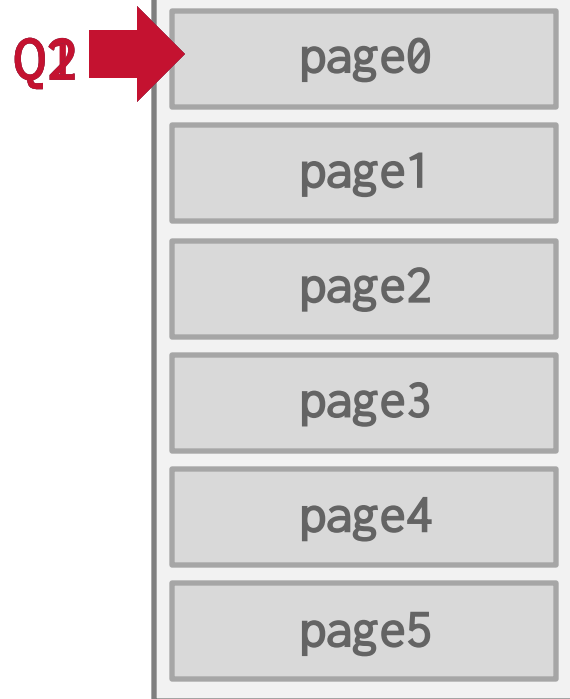
**Q2** SELECT AVG(val) FROM A

**Q3** SELECT \* FROM A WHERE id = 1

## *Buffer Pool*



## *Disk Pages*



# Better Policies: LRU-K

Track the history of last  $K$  references to each page as timestamps and compute the interval between subsequent accesses.

→ Can distinguish between reference types

Use this history to estimate the next time that page is going to be accessed.

→ Replace the page with the oldest "K-th" access.

→ Balances recency vs. frequency of access.

→ Maintain an ephemeral in-memory cache for recently evicted pages to prevent them from always being evicted.

## The LRU-K Page Replacement Algorithm For Database Disk Buffering

Elizabeth J. O'Neill<sup>1</sup>, Patrick E. O'Neill<sup>1</sup>, Gerhard Weikum<sup>2</sup>

<sup>1</sup> Department of Mathematics and Computer Science  
University of Massachusetts at Boston  
Harbor Campus  
Boston, MA 02125-3393

<sup>2</sup> Department of Computer Science  
ETH Zurich  
CH-8092 Zurich  
Switzerland

E-mail: ooneil@cs.umb.edu, pooneil@cs.umb.edu, weikum@inf.ethz.ch

### ABSTRACT

This paper introduces a new approach to database disk buffering, called the LRU-K method. The basic idea of LRU-K is to keep track of the times of the last  $K$  references to popular database pages, using this information to statistically estimate the interarrival times of references on a page by page basis. Although the LRU-K approach performs optimal statistical inference under relatively standard assumptions, it is fairly simple and incurs little bookkeeping overhead. As we demonstrate with simulation experiments, the LRU-K algorithm surpasses conventional buffering algorithms in discriminating between frequently and infrequently referenced pages. In fact, LRU-K can approach the behavior of buffering algorithms in which page sets with known access frequencies are manually assigned to different buffer pools of specifically tuned sizes. Unlike such customized buffering algorithms however, the LRU-K method is self-tuning, and does not rely on external hints about workload characteristics. Furthermore, the LRU-K algorithm adapts in real time to changing patterns of access.

### 1. Introduction

#### 1.1 Problem Statement

All database systems retain disk pages in memory buffers for a period of time after they have been read in from disk and accessed by a particular application. The purpose is to keep popular page memory resident and reduce disk I/O. In their "Five Minute Rule", Gray and Putzolu pose the following tradeoff: "We are willing to pay more for memory buffers up to a certain point, in order to reduce the cost of disk arms for a system (GRAYPUT), see also [CKS]. The critical buffering decision arises when a new buffer slot is needed for a page about to be read in from disk, and all current buffers are in use. What current page should be dropped from buffer? This is known as the page replacement policy, and the different buffering algorithms take their names from the type of replacement policy they impose (see, for example, [CORFENN], [BFFHHER]).

<sup>1</sup>Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, to republish, requires a fee and/or special permission.

SIGMOD '93/Washington, DC, USA  
© 1993 ACM 0-89791-592-6/93/00050297...11.50

The algorithm utilized by almost all commercial systems is known as LRU, for Least Recently Used. When a new buffer is needed, the LRU policy drops the page from buffer that has not been accessed for the longest time. LRU buffering was developed originally for patterns of use in instruction logs (for example, [DORNN], [CORFENN]), and does not always fit well into the database environment, as was noted also in [REITER], [STON], [SACSCH], and [CHODROW]. In fact, the LRU buffering algorithm has a problem which is addressed by the current paper: that it decides what page to drop from buffer based on too little information, limiting itself to only the time of last reference. Specifically, LRU is unable to differentiate between pages with relatively frequent references and pages that have very infrequent references until the system has wasted a lot of resources keeping infrequently referenced pages in buffer for an extended period.

**Example 1.1.** Consider a multi-user database application, which references randomly chosen customer records through a clustered B-tree indexed key, CUST\_ID, to retrieve desired information (cf. [TPC-A]). Assume simplistically that 20,000 customers exist, that a customer record is 2000 bytes in length, and that space needed for the B-tree index at the leaf level (five pages included, is 20 bytes for each key entry. Then if disk pages contain 4000 bytes of usable space and can be packed full, we require 100 pages to hold the leaf level nodes of the B-tree index (there is a single B-tree root node), and 10,000 pages to hold the records. The pattern of reference to these pages (ignoring the B-tree root node) is clearly: 11, R1, 12, R2, 13, R3, ..., alternate references to random index leaf pages and record pages. If we can only afford to buffer 100 pages in memory for this application, the B-tree root node is automatic; we should buffer all of the B-tree leaf pages, since each of them is referenced with a probability of .005 (once in each 200 general page reference), while it is clearly wasteful to displace one of these leaf pages with a data page, since data pages have only .0005 probability of reference (once in each 20,000 general page reference). Using the LRU algorithm, however, the pages held in memory buffers will be the hundred most recently referenced ones. To a first approximation, this means 50 B-tree leaf pages and 50 record pages. Given that a page gets no extra credit for being referenced twice in the recent past and that this is more likely to happen with B-tree leaf pages, there will even be slightly more data

297

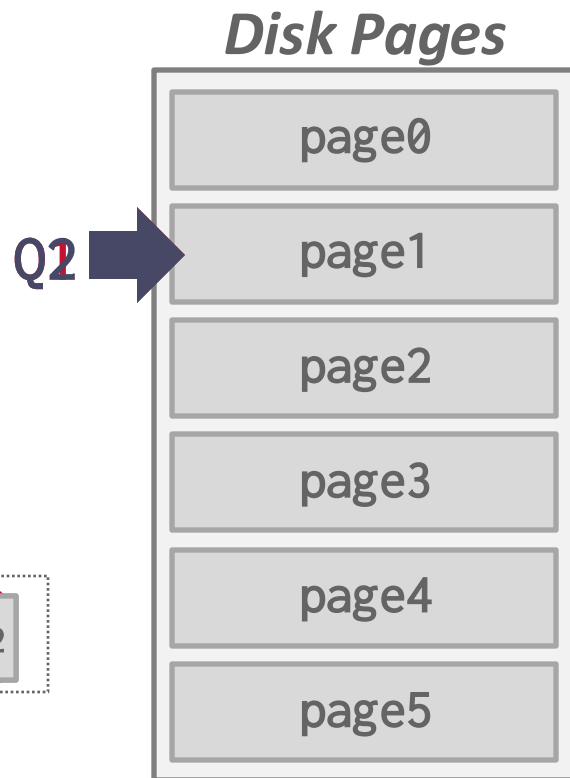
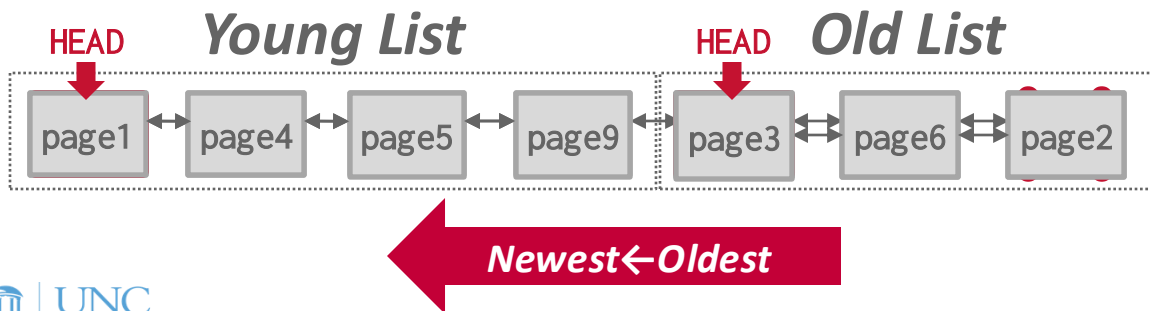




# MySQL: Approximate LRU-K

Single LRU linked list but with two entry points ("old" vs "young").

- New pages are always inserted to the head of the old list.
- If pages in the old list is accessed again, then insert into the head of the young list.



# Better Policies: Localization

The DBMS chooses which pages to evict on a per query basis. This minimizes the pollution of the buffer pool from each query.

→ Keep track of the pages that a query has accessed.

Example: Postgres assigns a limited number of buffer pool pages to a query and uses it as a circular ring buffer.

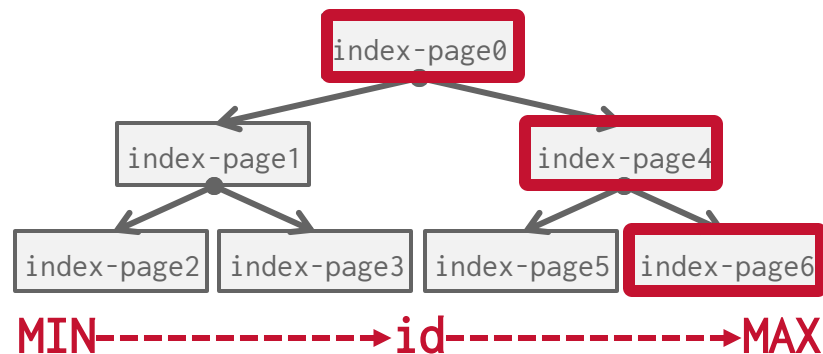
# Better Policies: Priority Hints

The DBMS knows about the context of each page during query execution.

It can provide hints to the buffer pool on whether a page is important or not.

Q1 INSERT INTO A VALUES (*id++*)

Q2 SELECT \* FROM A WHERE id = ?



# Dirty Pages

**Fast Path:** If a page in the buffer pool is not dirty, then the DBMS can simply "drop" it.

**Slow Path:** If a page is dirty, then the DBMS must write back to disk to ensure that its changes are persisted.

Trade-off between fast evictions versus writing dirty pages that will not be read again in the future.

# Background Writing

The DBMS can periodically walk through the page table and write dirty pages to disk.

When a dirty page is safely written, the DBMS can either evict the page or just unset the dirty flag.

Need to be careful that the system writes dirty pages in a safe order

- Need to be able to recover from a crash
- How would I transfer \$\$\$ from one account to another?

# Observation

OS/hardware tries to maximize disk bandwidth by reordering and batching I/O requests.

But they do not know which I/O requests are more important than others.

Many DBMSs tell you to switch Linux to use the deadline or noop (FIFO) scheduler.

→ Example: Oracle, Vertica, MySQL

# Disk I/O Scheduling

The DBMS maintains internal queue(s) to track page read/write requests from the entire system.

Compute priorities based on several factors:


- Sequential vs. Random I/O
- Critical Path Task vs. Background Task
- Table vs. Index vs. Log vs. Ephemeral Data
- Transaction Information
- User's performance targets

The OS doesn't know these things and is going to get into the way...

Most disk operations go through the OS API. Unless the DBMS tells it not to, the OS maintains its own filesystem cache (aka page cache, buffer cache).

Most DBMSs use direct I/O (O\_DIRECT) to bypass the OS's cache

- Redundant copies of pages.
- Different eviction policies.
- Loss of control over file I/O.



**Krishnakumar R** • 3rd+  
 Group Engineering Manager, PostgreSQL engine @ MicroS...  
 4mo •

[+ Follow](#)
...

### Direct IO in PostgreSQL and double buffering

The following was an experiment I had shown in my talk on PostgreSQL and Kernel interactions at PGDay Chicago last week :-)

The left side shows the default setting. When contents from a table are read, it will get cached both in the postgres buffer pool and kernel page cache. The third command shows the page details from the pg buffer pool, and the last command (uses fcore utility) shows info on how much the file corresponding to the table (refresh note: PostgreSQL uses files for its data storage) is cached in the kernel. Note that PG has 8K block size while Kernel has 4K pages (x64 in this case).

On the right you can see developer debug setting which is present from PG16 onwards for enabling direct io is switched on for 'data'. This results in the pages no longer cached in kernel page cache and only cached in buffer pool of pg. As resultant you can see from the output from fcore not pages are cached in page cache.

**#postgres #PostgreSQL #Kernel #PageCache #Linux #LinuxKernel**

```

postgres=# show debug_io_direct;
debug_io_direct
(1 row)

postgres=# select * from map limit 10;
 i | t
---+---
 1 | 
200 | Two Hundred
300 | Three Hundred
400 | Four Hundred
500 | Five Hundred
600 | Six Hundred
700 | Seven Hundred
800 | Eight Hundred
900 | Nine Hundred
1000 | Thousand
(10 rows)

postgres=# select bufferid,relfilnode from pg_buffercache where relfilnode=16384;
 bufferid | relfilnode
-----+-----
        98 |      16384
(1 row)

postgres=# \! fcore install-pg/data/base/5/16384
PGS SIZE FILE
0 0K Install-pg/data/base/5/16384

```

```

postgres=# show debug_io_direct;
debug_io_direct
(1 row)

postgres=# select * from map limit 10;
 i | t
---+---
 100 | Hundred
200 | Two Hundred
300 | Three Hundred
400 | Four Hundred
500 | Five Hundred
600 | Six Hundred
700 | Seven Hundred
800 | Eight Hundred
900 | Nine Hundred
1000 | Thousand
(10 rows)

postgres=# select bufferid,relfilnode from pg_buffercache where relfilnode=16384;
 bufferid | relfilnode
-----+-----
        98 |      16384
(1 row)

postgres=# \! fcore install-pg/data/base/5/16384
PGS PAGES SIZE FILE
0 0K Install-pg/data/base/5/16384

```



# Buffer Pool Optimizations

Multiple Buffer Pools

Pre-Fetching

Scan Sharing

Buffer Pool Bypass

# Multiple Buffer Pools

The DBMS does not always have a single buffer pool for the entire system.

- Multiple buffer pool instances
- Per-database buffer pool
- Per-page type buffer pool

Partitioning memory across multiple pools helps reduce latch contention and improve locality.

- Avoids contention on LRU tracking metadata.



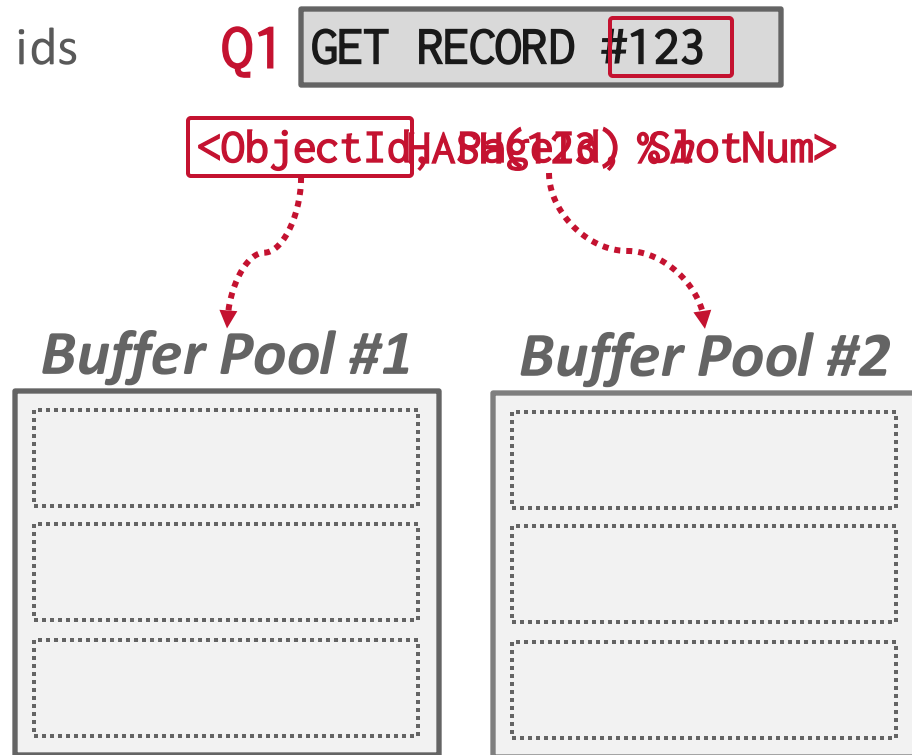
# Multiple Buffer Pools

## Approach #1: Object Id

→ Embed an object identifier in record ids and then maintain a mapping from objects to specific buffer pools.

## Approach #2: Hashing

→ Hash the page id to select which buffer pool to access.

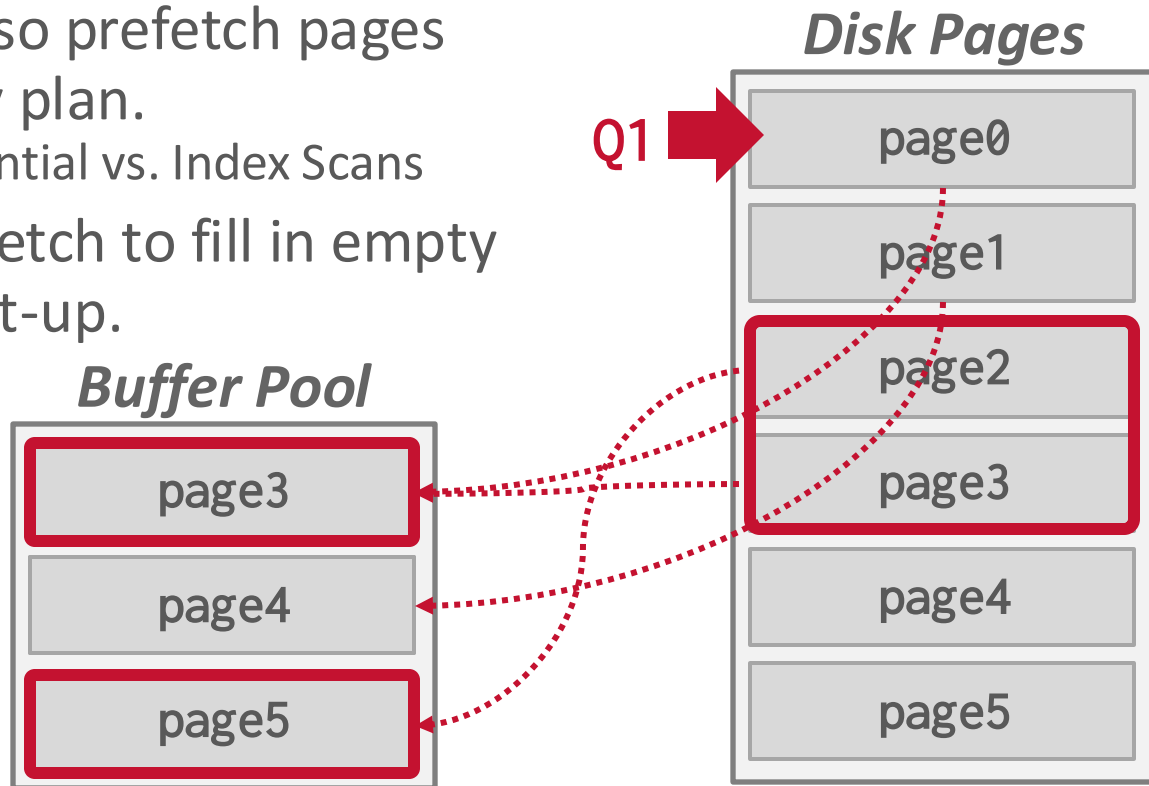


# Pre-Fetching

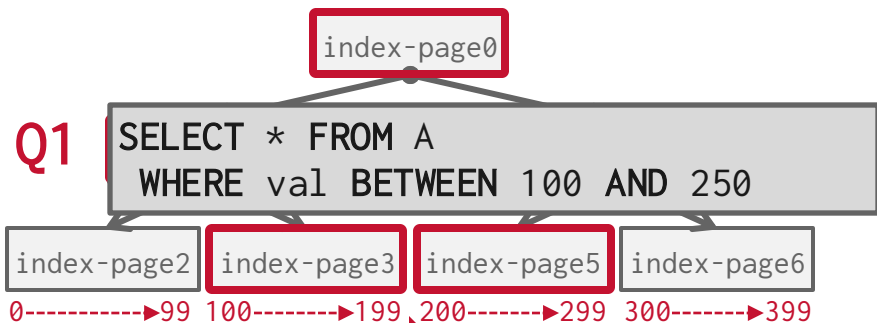
The DBMS can also prefetch pages based on a query plan.

→ Examples: Sequential vs. Index Scans

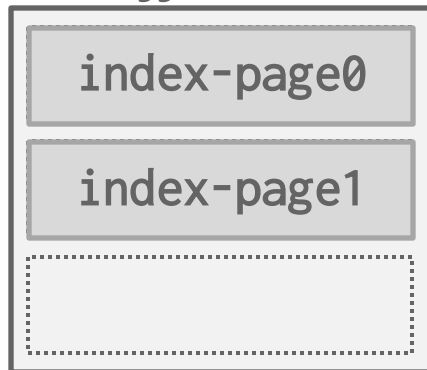
Some DBMS prefetch to fill in empty frames upon start-up.



# Pre-Fetching



## Buffer Pool



## Disk Pages



# Scan Sharing


Allow multiple queries to attach to a single cursor that scans a table.

→ Also called *synchronized scans*.

→ This is different from result caching.

For a textual match to occur, the text of the SQL statements or PL/SQL blocks must be character-for-character identical, including spaces, case, and comments. For example, the following statements cannot use the same shared SQL area:

```
SELECT * FROM employees;  
SELECT * FROM Employees;  
SELECT * FROM employees;
```

 Copy

ORACLE® reSQL

# Scan Sharing

**Q1** `SELECT SUM(val) FROM A`

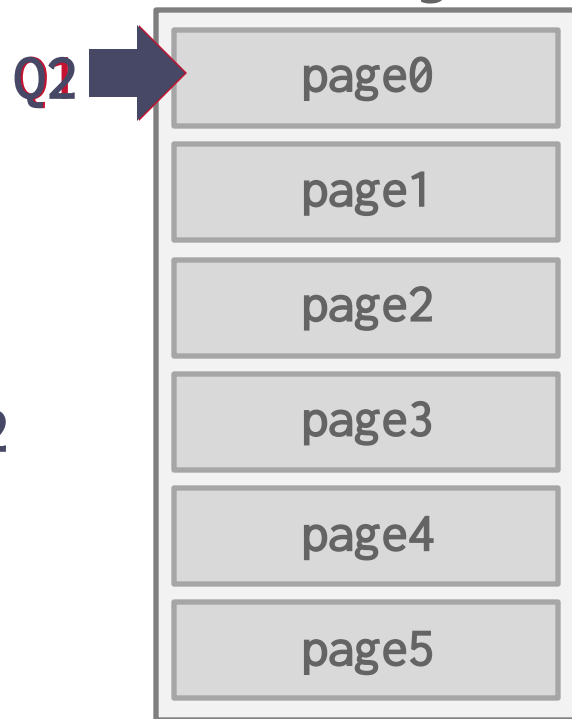
**Q2** `SELECT AVG(val) FROM A LIMIT 100`

## *Buffer Pool*



**Q2**

## *Disk Pages*



# Buffer Pool Bypass

The sequential scan operator will not store fetched pages in the buffer pool to avoid overhead.

- Memory is local to running query.
- Works well if operator needs to read a large sequence of pages that are contiguous on disk.
- Can also be used for temporary data (sorting, joins).

Called "Light Scans" in Informix.

ORACLE®

Microsoft®  
SQL Server®

Informix®



# Conclusion

The DBMS can almost always manage memory better than the OS.

Leverage the semantics about the query plan to make better decisions:

- Evictions
- Allocations
- Pre-fetching

# Next Class

Wrapping up storage...

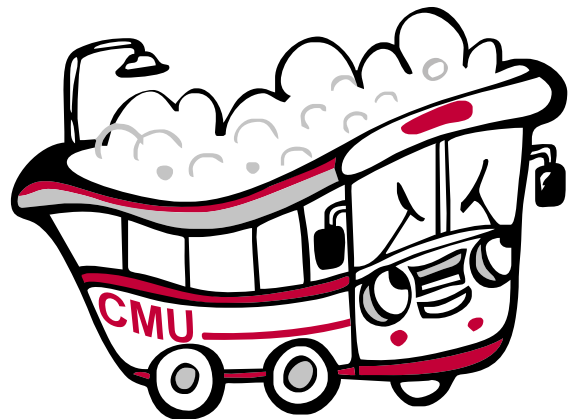
Column stores and compression

# Project #1

You will build the first component of your storage manager.

- LRU-K Replacement Policy
- Disk Scheduler
- Buffer Pool Manager Instance

We will provide you with the basic APIs for these components.



## BusTub

**Due Date:**  
**Sunday Sept 29<sup>th</sup> @ 11:59pm**

# Task #1 – LRU-K Replacement Policy

Build a data structure that tracks the usage of pages using the LRU-K policy.

General Hints:

- Your **LRUKReplacer** needs to check the "pinned" status of a **Page**.
- If there are no pages touched since last sweep, then return the lowest page id.

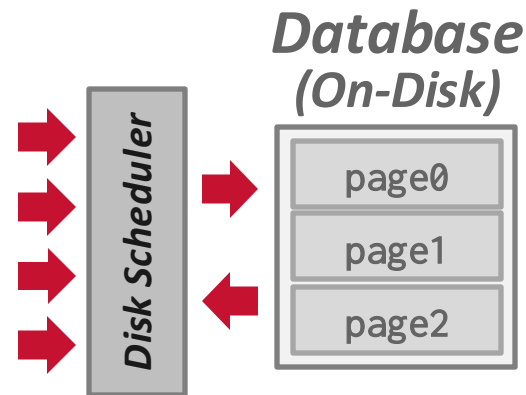
## Task #2 – Disk Scheduler

Create a background worker to read/write pages from disk.

- Single request queue.
- Simulates asynchronous IO using **`std::promise`** for callbacks.

It's up to you to decide how you want to batch, reorder, and issue read/write requests to the local disk.

Make sure it is thread-safe!

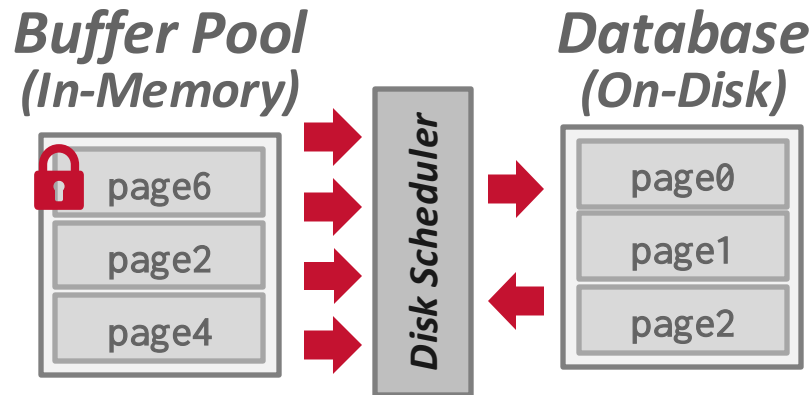


# Task #3 – Buffer Pool Manager

Use your LRU-K replacer to manage the allocation of pages.

- Need to maintain internal data structures to track allocated + free pages.
- Implement page guards.
- Use whatever data structure you want for the page table.

Make sure you get the order of operations correct when pinning!



# Things To Note

Do not change any file other than the six that you must hand in. Other changes will not be graded.

The projects are cumulative.

We will not be providing solutions.

Come to office hours for high-level questions, but we will not help you debug.

# Code Quality

We will automatically check whether you are writing good code.

- [Google C++ Style Guide](#)
- [Doxygen Javadoc Style](#)

You need to run these targets before you submit your implementation to Gradescope.

- `make format`
- `make check-clang-tidy-p1`



# Extra Credit

Gradescope Leaderboard runs your code with a specialized in-memory version of BusTub.

The top 20 fastest implementations in the class will receive extra credit for this assignment.

- **#1**: 50% bonus points
- **#2–10**: 25% bonus points
- **#11–20**: 10% bonus points

Student with the most bonus points at the end of the semester will get some prize TBD