Carnegie Mellon University

DATABASE SYSTEMS

Buffer Pools

LECTURE #04 » 15-445/645 FALL 2025 » PROF. ANDY PAVLO

ADMINISTRIVIA

~~**Project #0** is due Sunday Sept 7th @ 11:59pm~~

~~**Homework #1** is due Sunday Sept 7th @ 11:59pm~~

Project #1 is due Sunday Sept 29th @ 11:59pm

UPCOMING DATABASE EVENTS

CMU-DB Industry Affiliates Visit Day

→ Monday Sept 15th: Research Talks + Poster Session
→ Tuesday Sept 16th: Company Info Sessions
→ All events are open to the public.

Carnegie Mellon University
Database Group
Industry Affiliates

Sign-up for Company Info Sessions (@54)

Add your Resume if You Want a Database Job (@55)

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.

← Today

DATABASE STORAGE

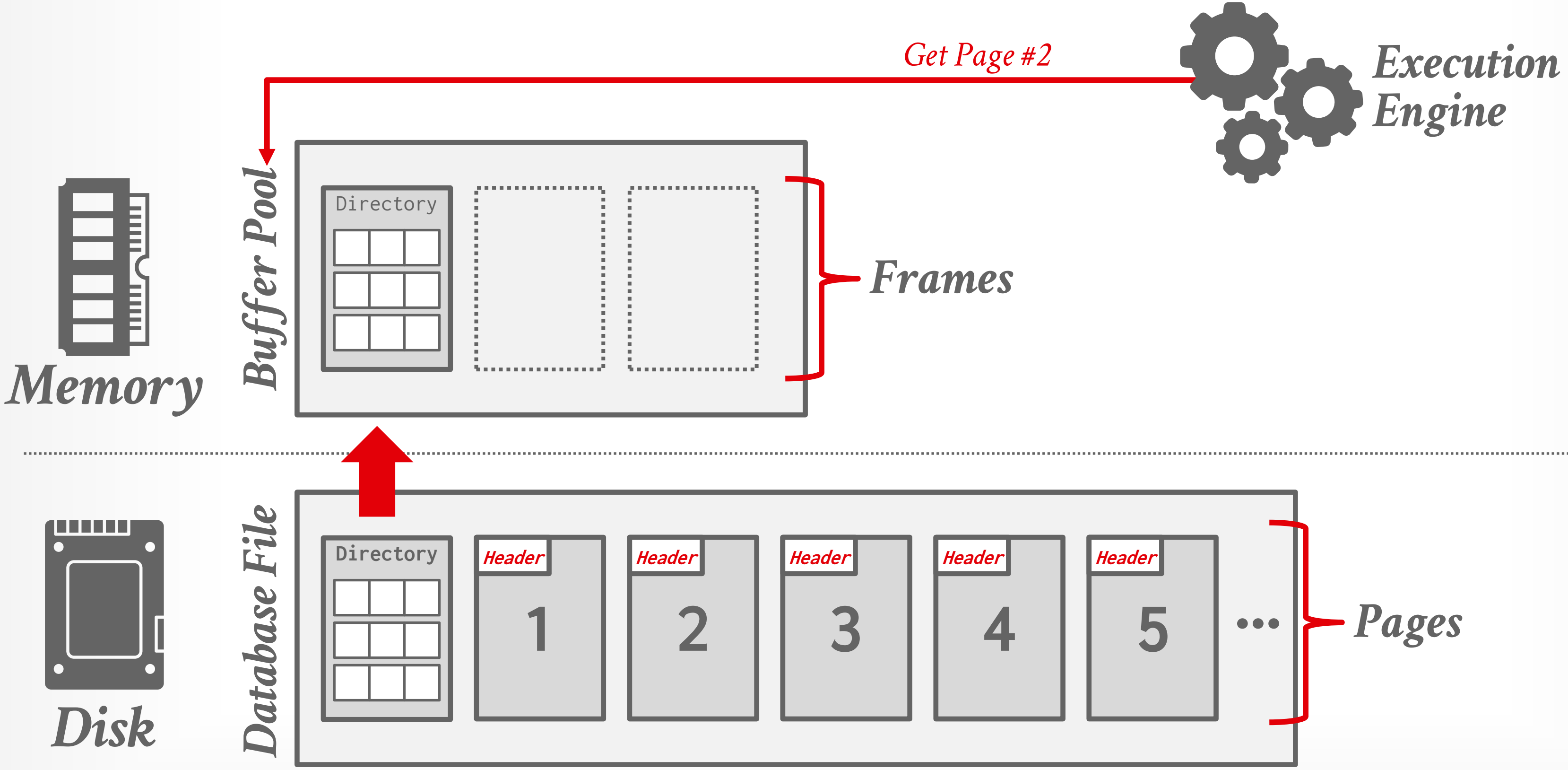
Spatial Control:

→ 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.

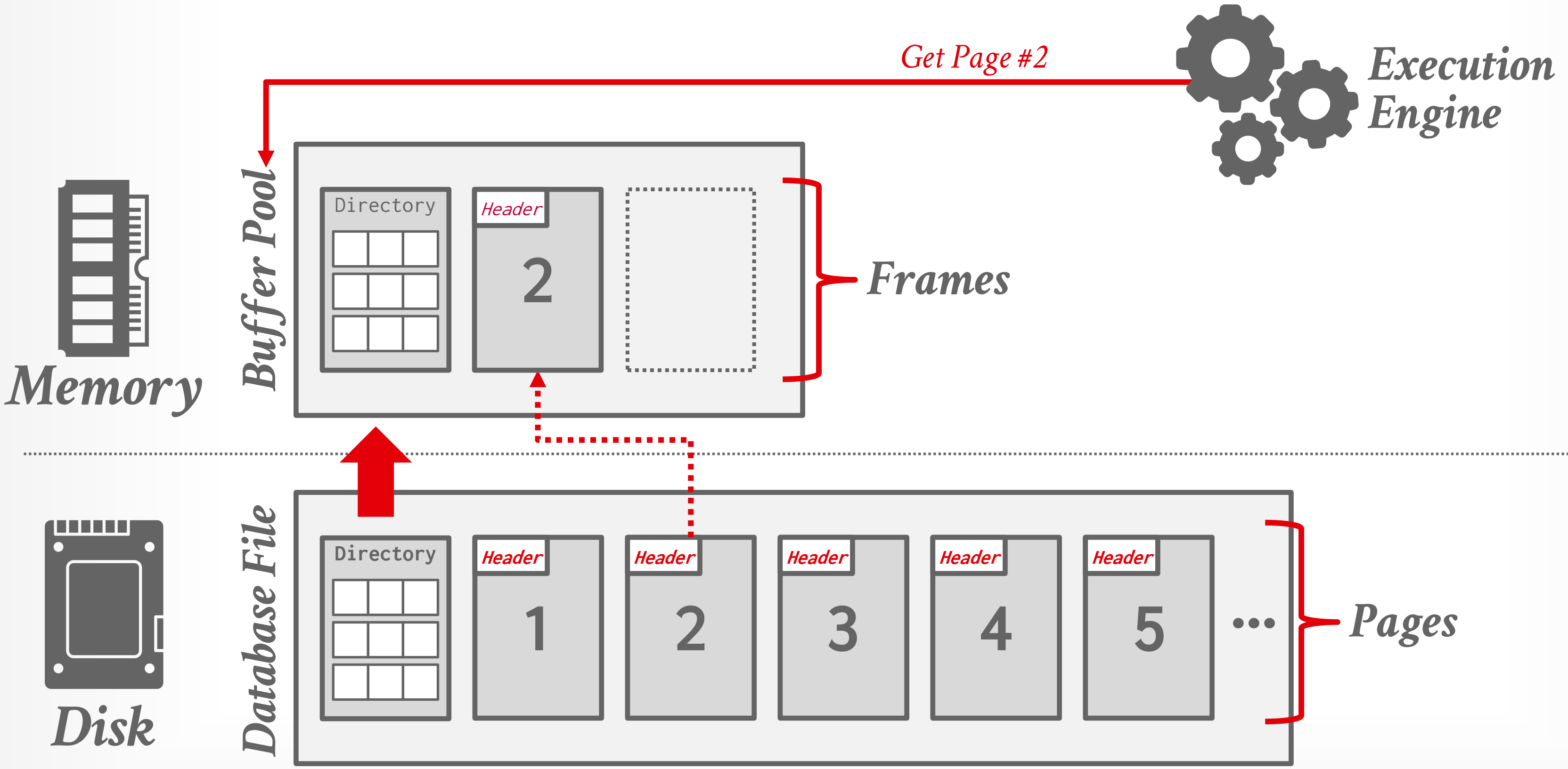
Temporal Control:

→ When to read pages into memory, and when to write them back to disk if they get changed.
→ The goal is to minimize the number of stalls from having to read data from disk.

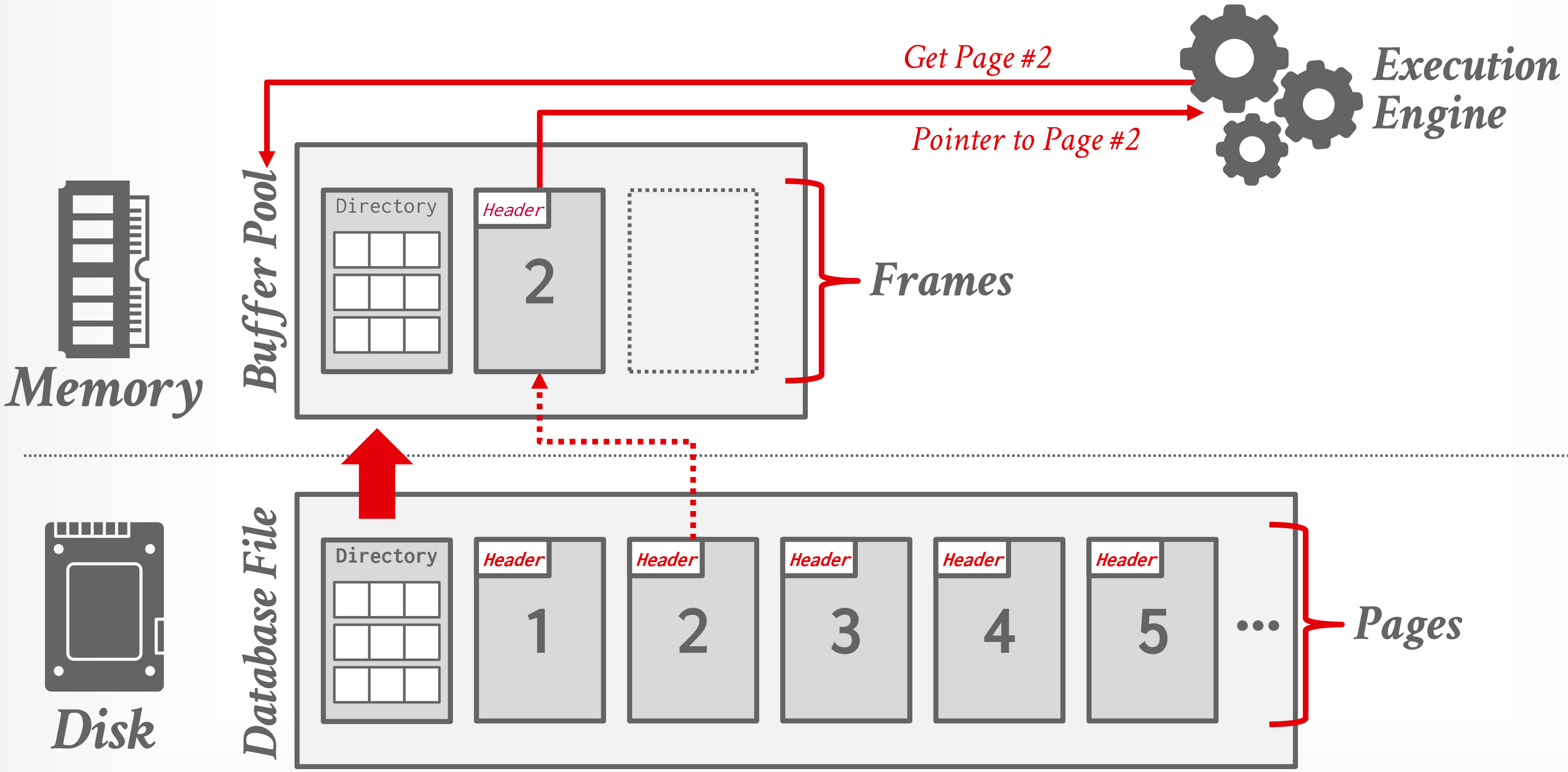
DISK-ORIENTED DBMS



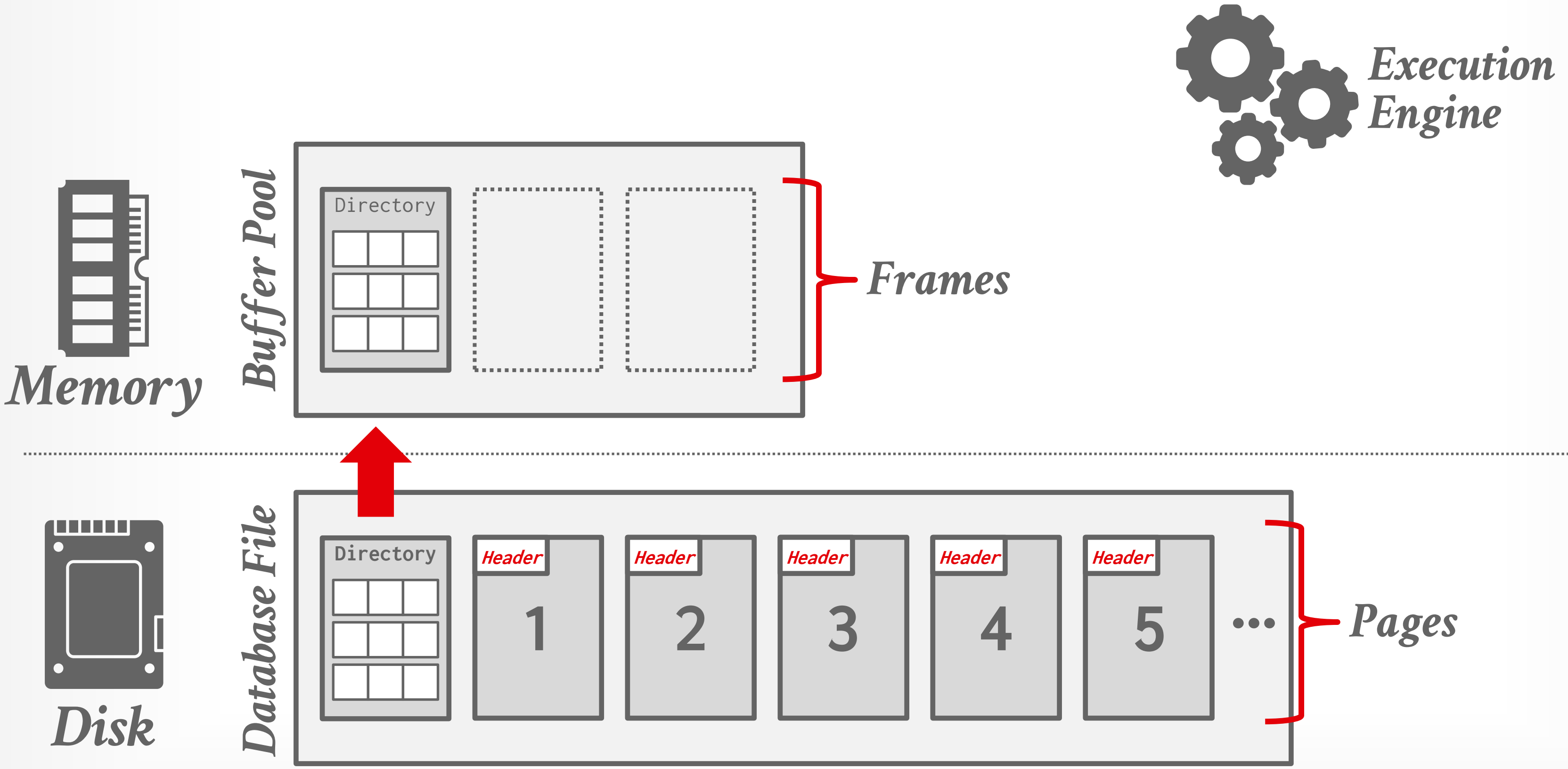
DISK-ORIENTED DBMS



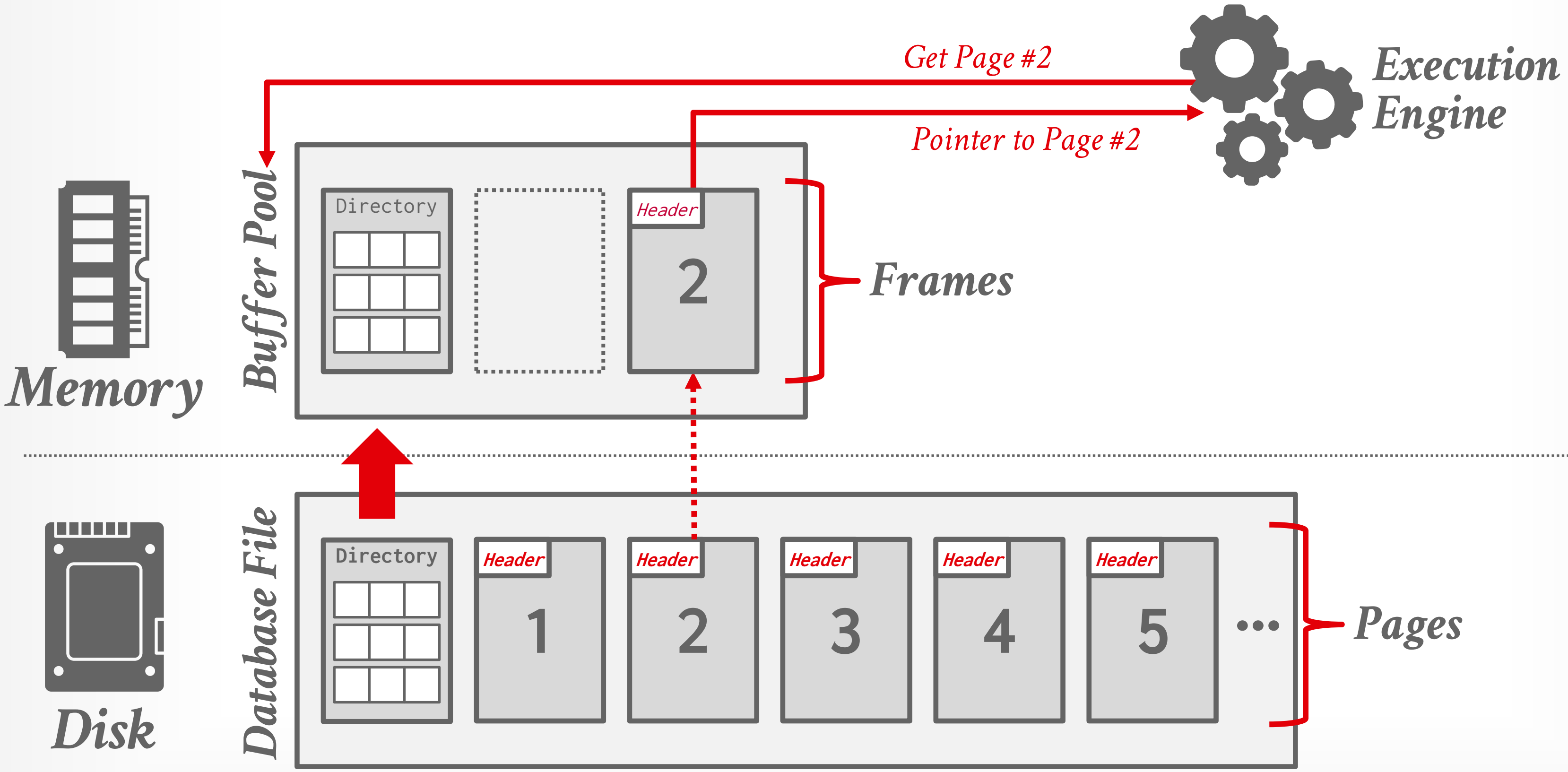
DISK-ORIENTED DBMS



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 backed by disk. Depends on implementation.

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

TODAY'S AGENDA

Buffer Pool Manager

Memory-Mapped Files?

Replacement Policies

Disk I/O Scheduling

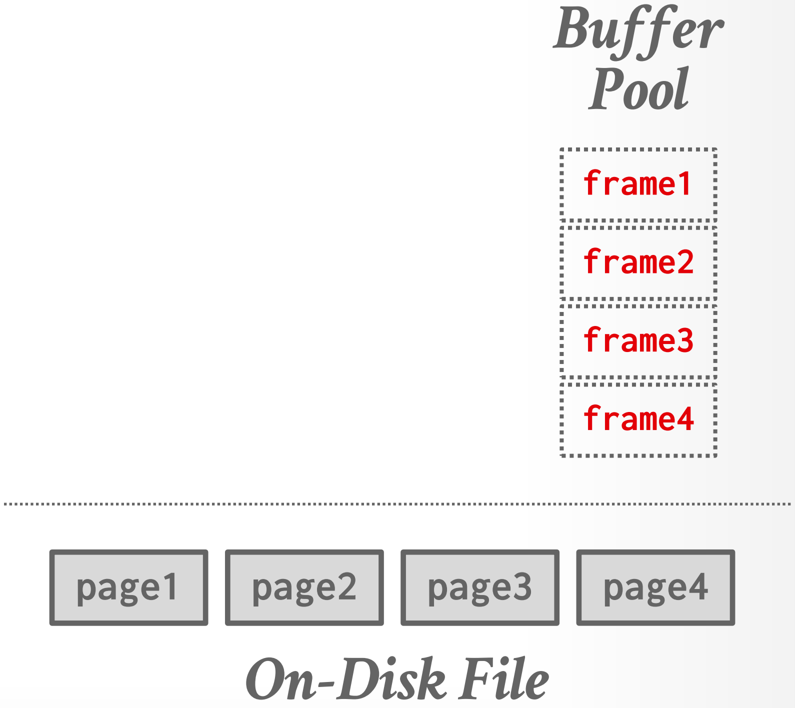
Optimizations

BUFFER POOL ORGANIZATION

Memory region organized as an array of fixed-size pages. Each array entry is called a <u>**frame**</u>.

When the DBMS requests a page, it places an exact copy of that page into one of these frames.

Dirty pages are buffered and <u>not</u> written to disk immediately
→ Write-Back Cache

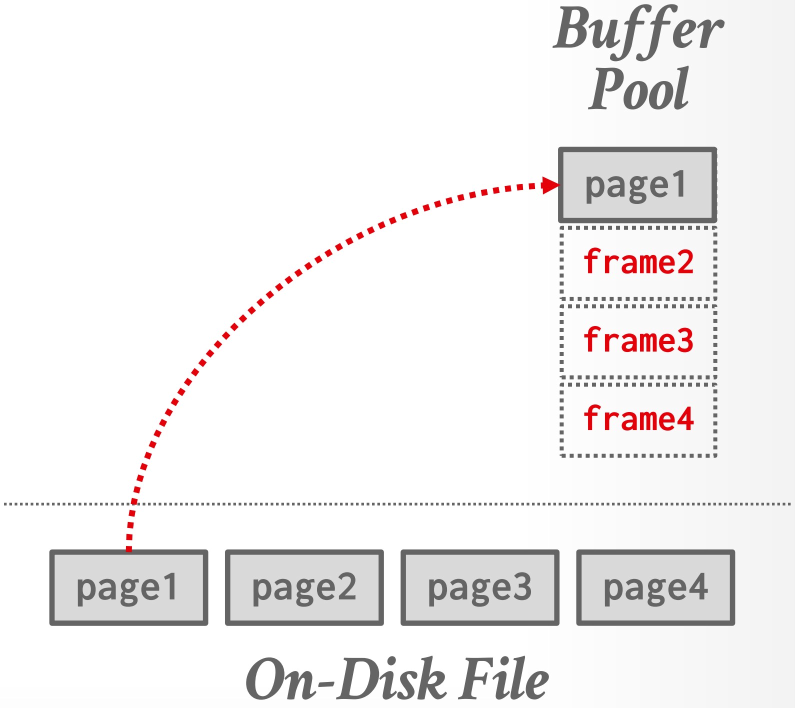


BUFFER POOL ORGANIZATION

Memory region organized as an array of fixed-size pages. Each array entry is called a **frame**.

When the DBMS requests a page, it places an exact copy of that page into one of these frames.

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

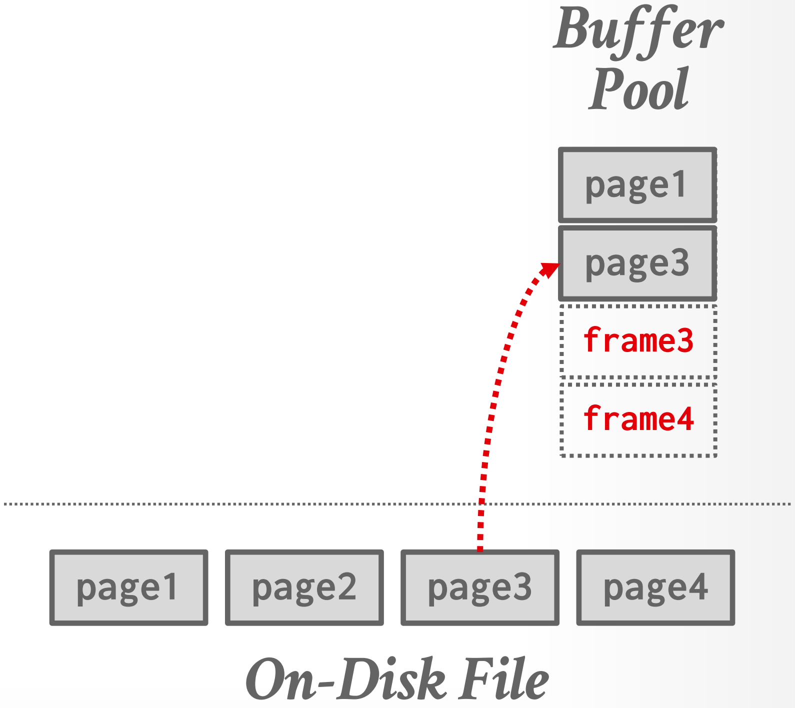


BUFFER POOL ORGANIZATION

Memory region organized as an array of fixed-size pages. Each array entry is called a **frame**.

When the DBMS requests a page, it places an exact copy of that page into one of these frames.

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



BUFFER POOL META-DATA

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

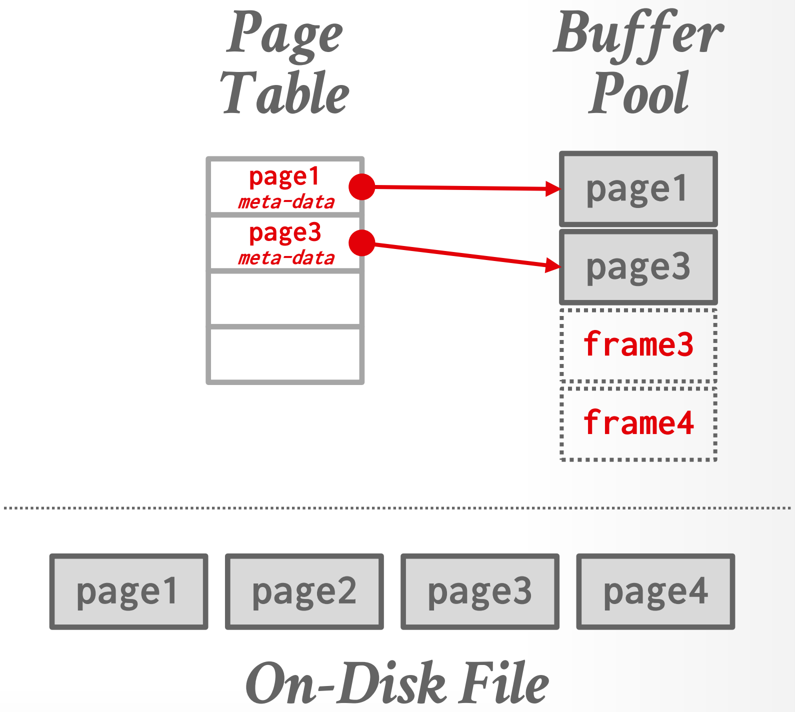
→ Usually a fixed-size hash table protected with latches to ensure thread-safe access.

Additional meta-data per page:

→ **Dirty Flag**

→ **Pin/Reference Counter**

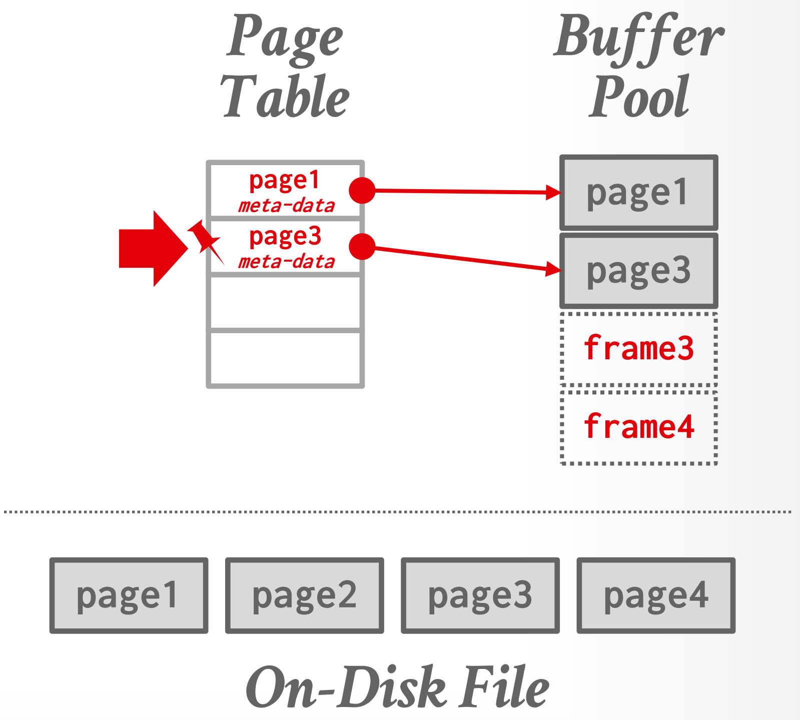
→ **Access Tracking Information**



BUFFER POOL META-DATA

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.

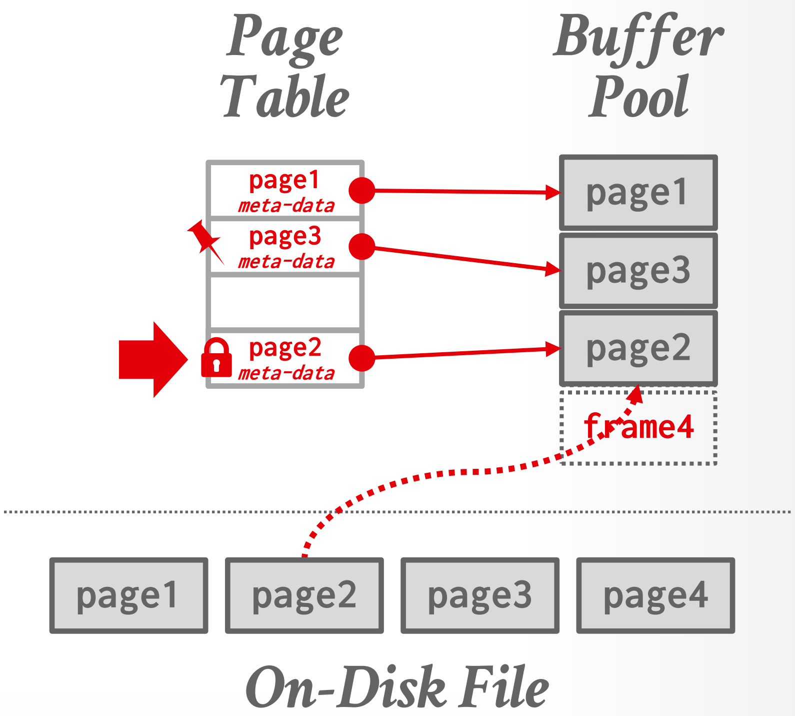
Additional meta-data per page:
→ **Dirty Flag**
→ **Pin/Reference Counter**
→ **Access Tracking Information**



BUFFER POOL META-DATA

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**



BUFFER POOL META-DATA

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

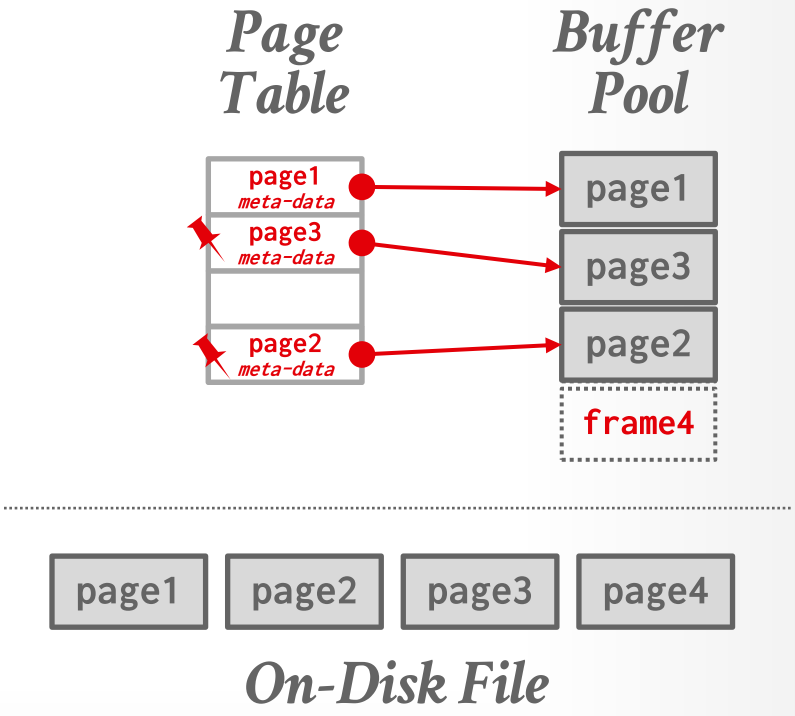
→ Usually a fixed-size hash table protected with latches to ensure thread-safe access.

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 structures from other workers (e.g., 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.

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?

page1	page2	page3	page4

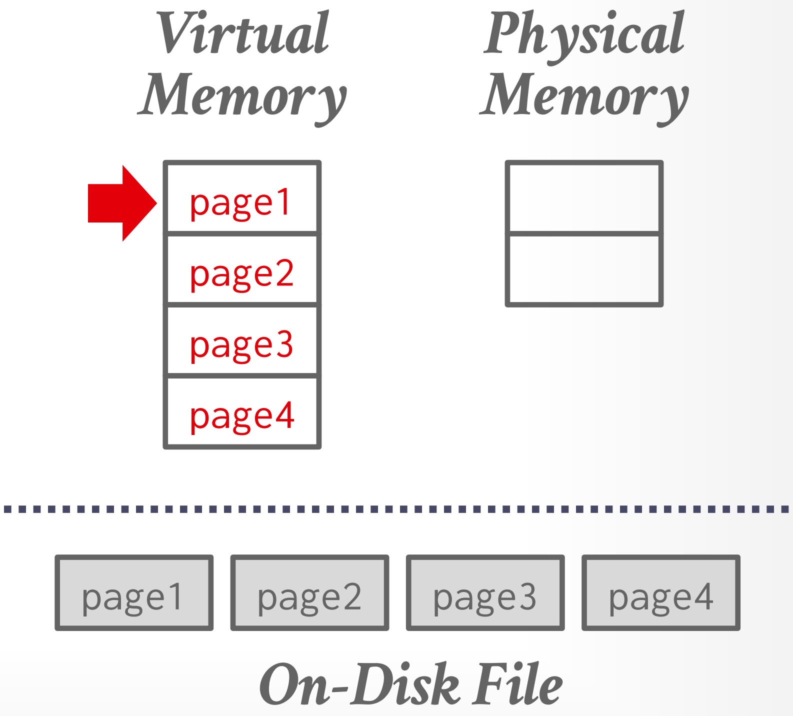
On-Disk File

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?

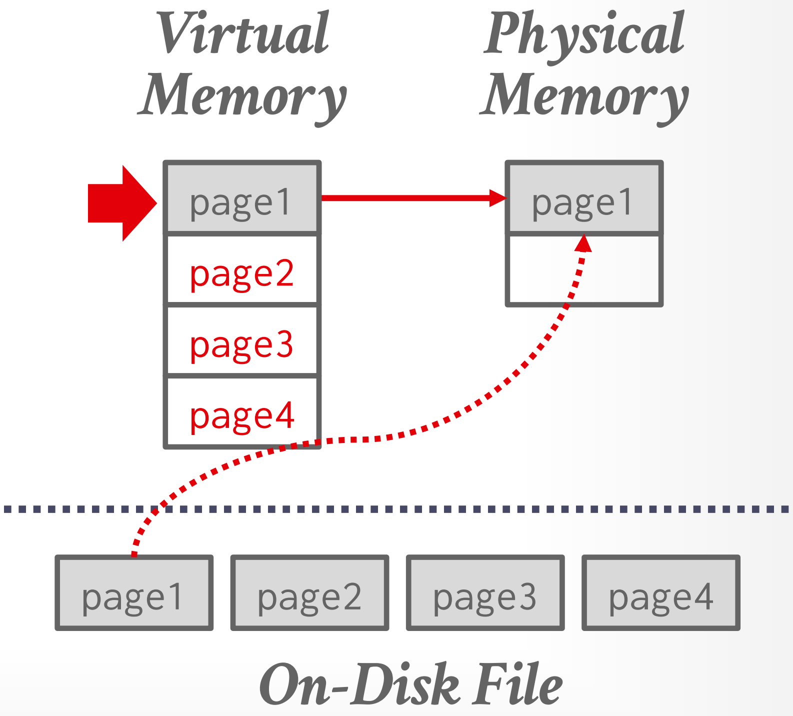


WHY NOT USE THE OS?

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?

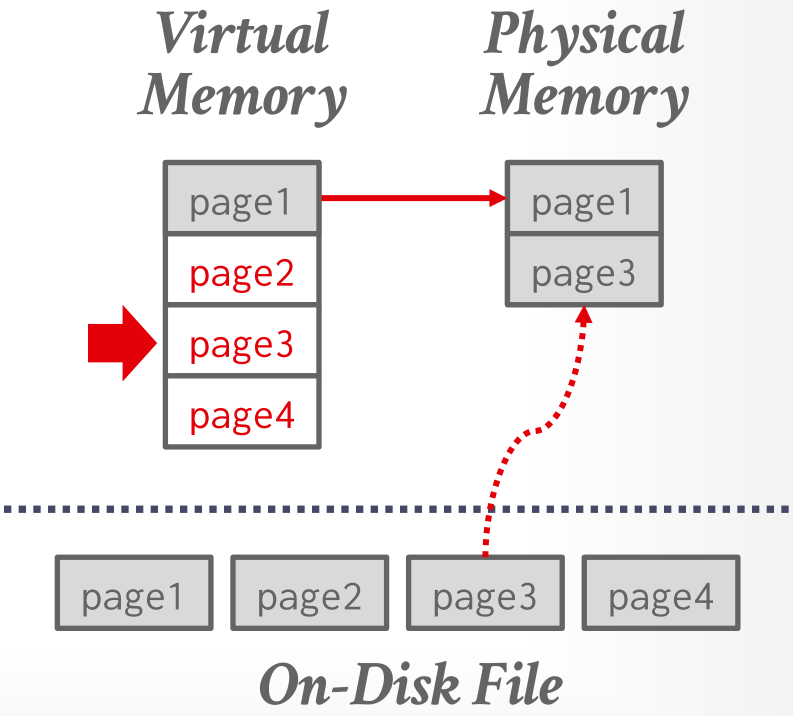


WHY NOT USE THE OS?

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?

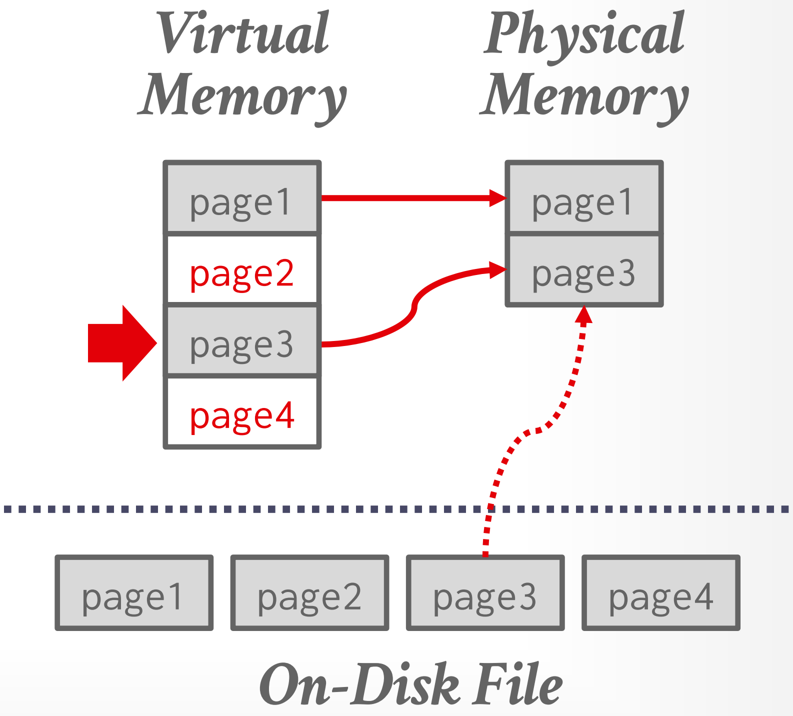


WHY NOT USE THE OS?

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?

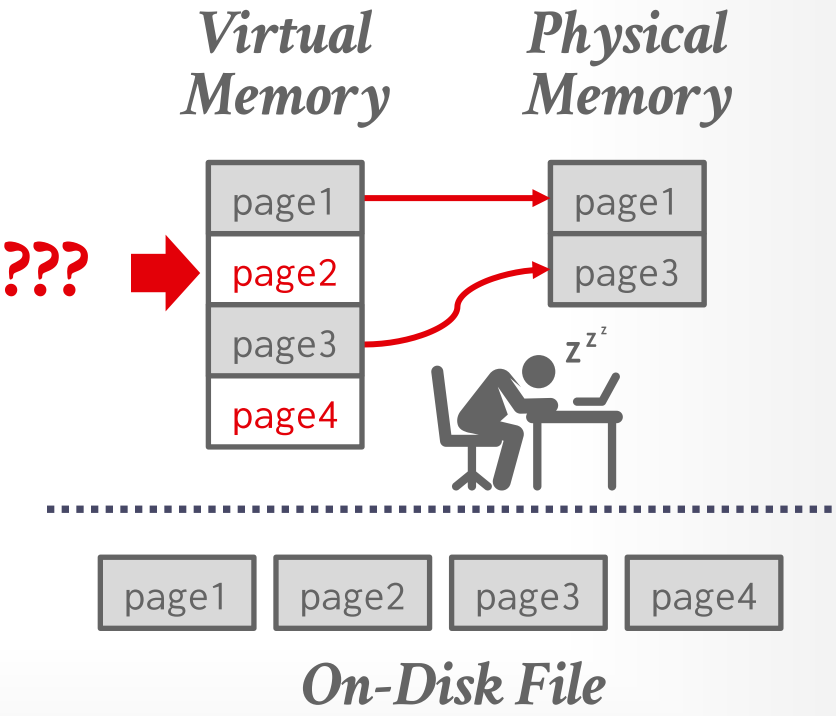


WHY NOT USE THE OS?

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.

P

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

Problem #4: Performance Issues

→ OS data structure contention. 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

monetdb LMDB levelDB

RavenDB QuestDB Weaviate

elasticsearch Yellowbrick MongoDB

Partial Usage

SingleStore WIRED TIGER

SQLite influxdb

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 THE OS?

DBMS (almost) always wants to control things itself and can do a better job than the OS.

→ Flushing dirty pages to disk in the correct order.
→ Specialized prefetching.
→ Buffer replacement policy.
→ Thread/process scheduling.

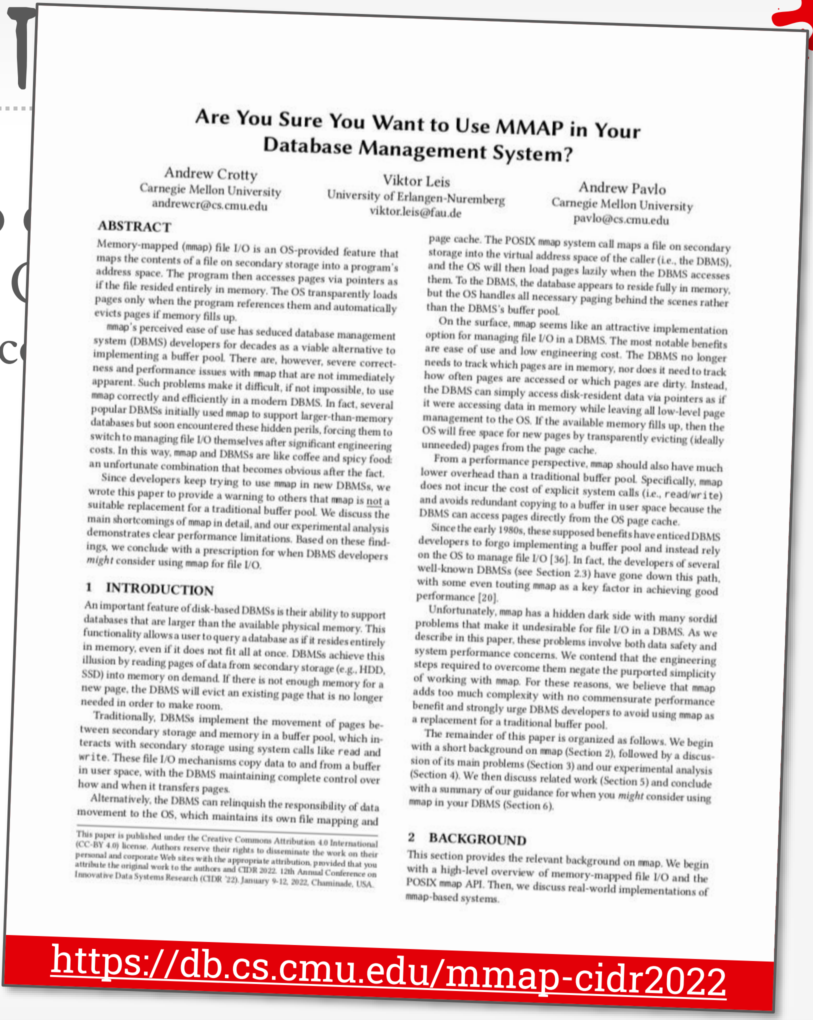
The OS is **not** your friend.

WHY NOT USE T

DBMS (almost) always wants to c
and can do a better job than the O

→ Flushing dirty pages to disk in the c
→ 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 resides entirely in memory, even if it does not fit all at once. 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).

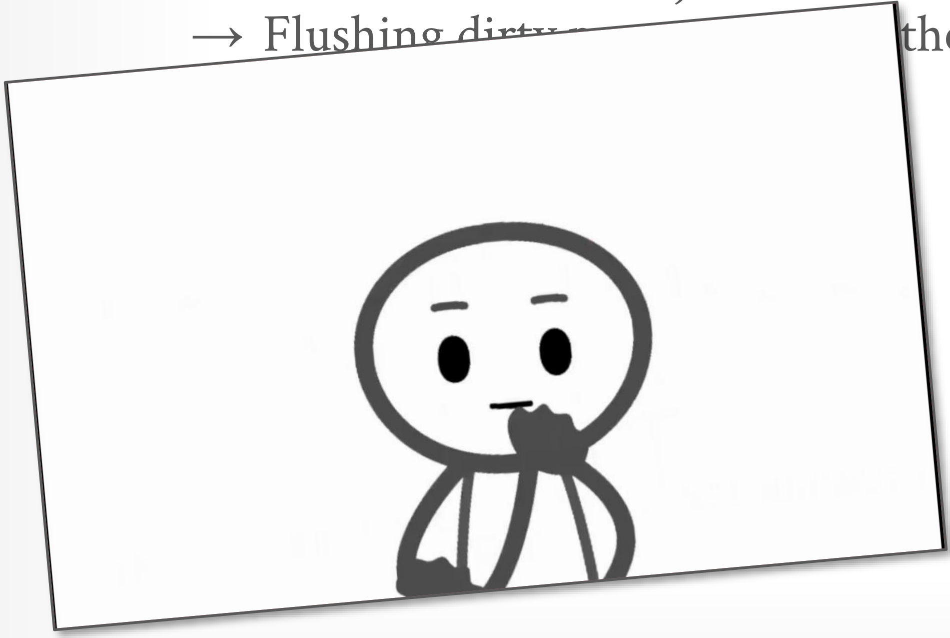
2 BACKGROUND

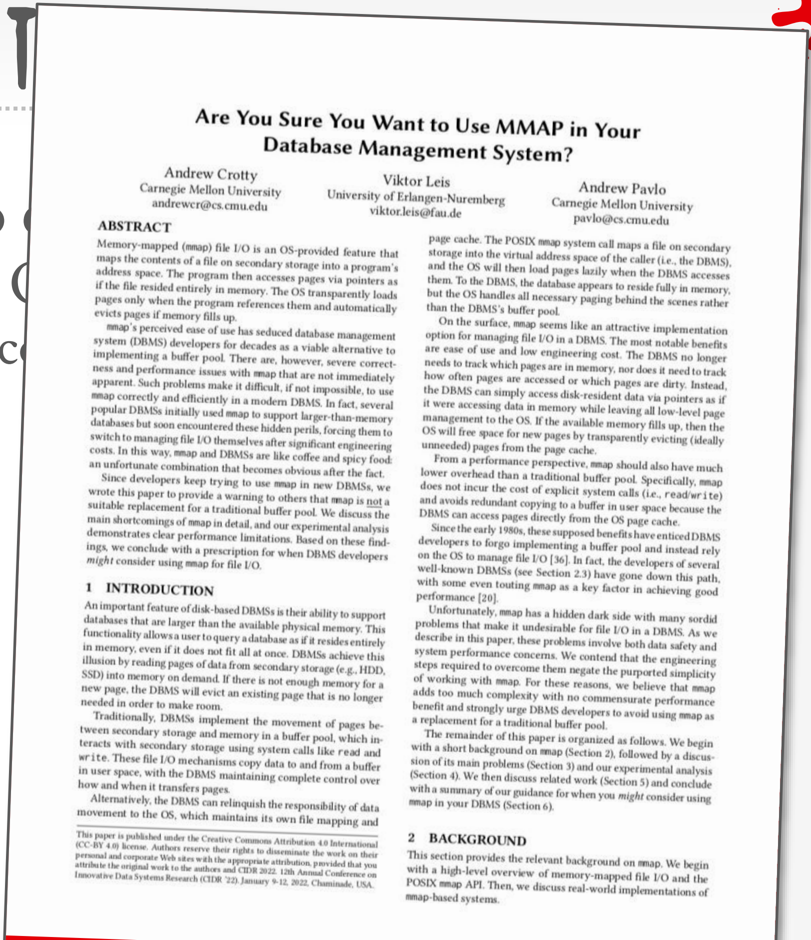
This section provides the relevant background on mmap. We begin with a high-level overview of memory-mapped file I/O and the POSIX mmap API. Then, we discuss real-world implementations of mmap-based systems.

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

WHY NOT USE T

DBMS (almost) always wants to
and can do a better job than the
→ Flushing dirty the c





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 resides entirely in memory, even if it does not fit all at once. 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).

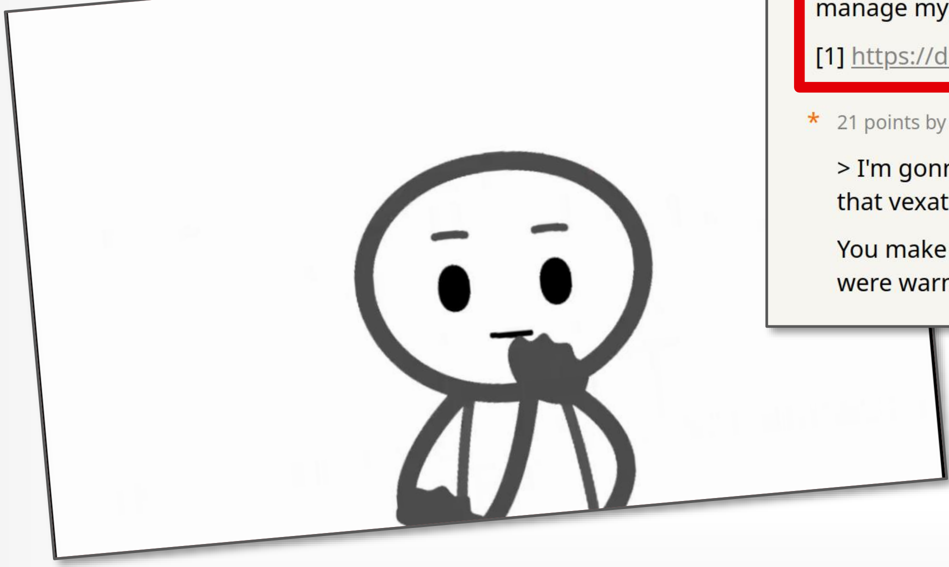
2 BACKGROUND

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

WHY NOT USE T

DBMS (almost) always
and can do a better job
→ Flushing dirty



▲ marginalia_nu 42 days ago | prev | next [–]

Optimizing the Marginalia Search index code. The new code is at least twice as fast in benchmarks, but I can't run it in production because it turns out when you do it's four times as slow as what came before it for the queries that are the simplest and fastest to the point where queries exceed their timeout values by a lot.

I'm 97% certain this is because the faster code leads to more page thrashing in the mmap-based index readers. I'm gonna have to implement my own buffer pool and manage my reads directly like that vexatious paper[1] said all along.

[1] https://db.cs.cmu.edu/papers...

* 21 points by apavlo 42 days ago | parent | next [–]

> I'm gonna have to implement my own buffer pool and manage my reads directly like that vexatious paper[1] said all along.

You make it sound like I was trying to troll everyone when we wrote that paper. We were warning you.

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

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

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.

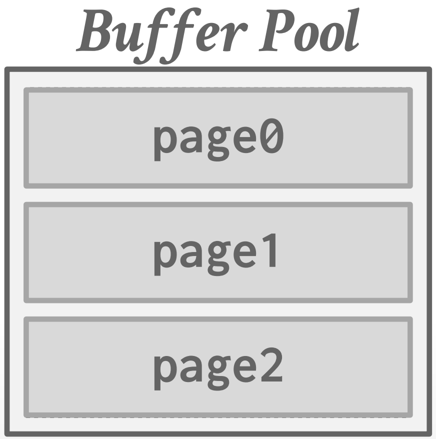
Goals:

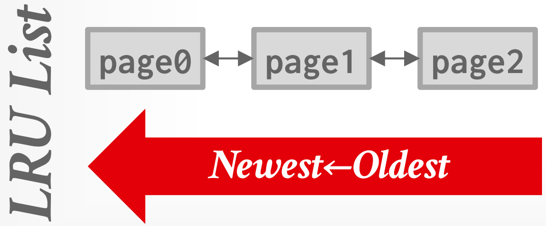
→ Correctness
→ Accuracy
→ Speed
→ Meta-data overhead

LEAST-RECENTLY USED (1965)

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.

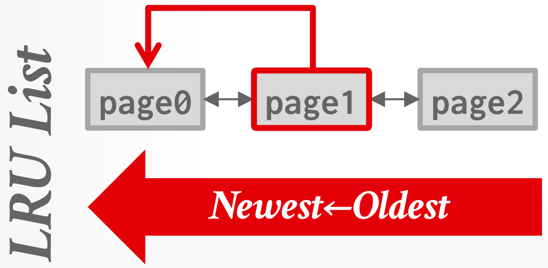


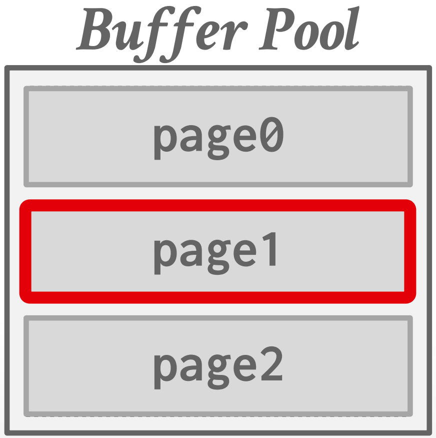


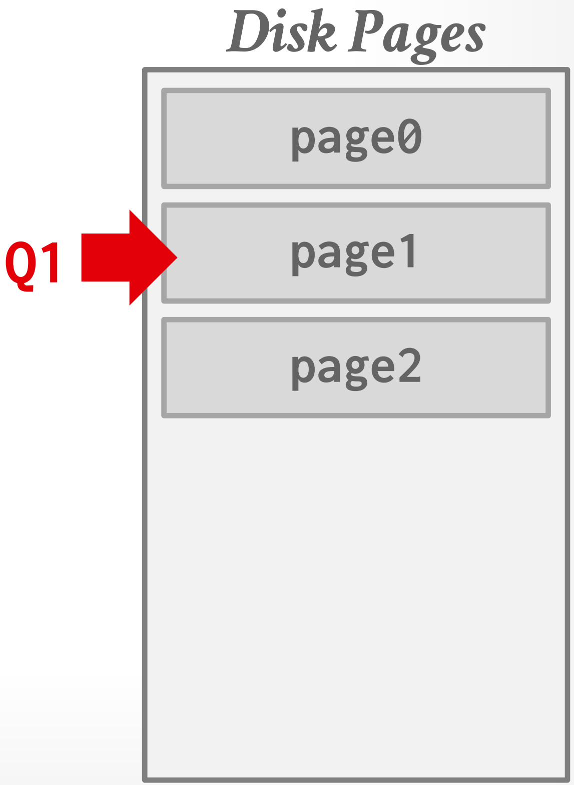
LEAST-RECENTLY USED (1965)

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.



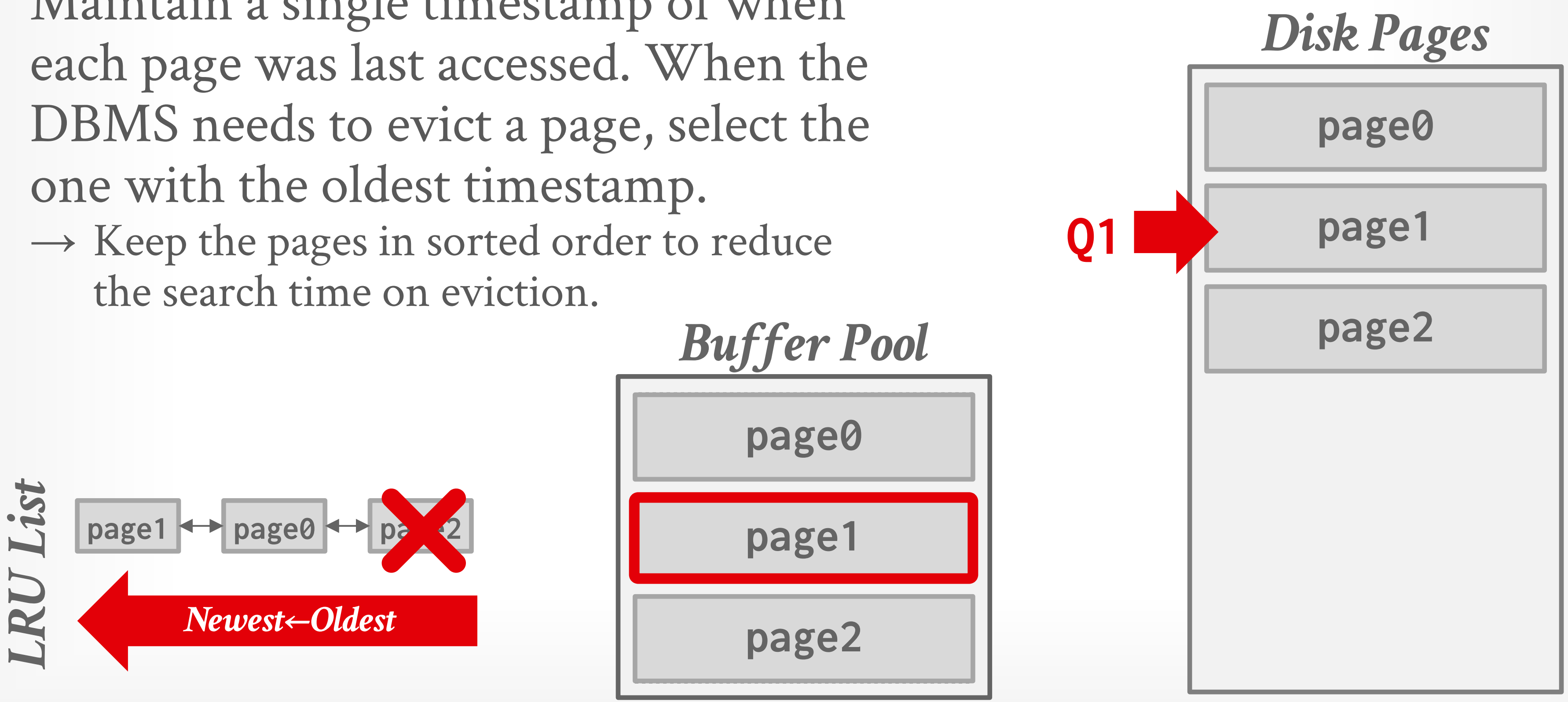




LEAST-RECENTLY USED (1965)

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 (1969)

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

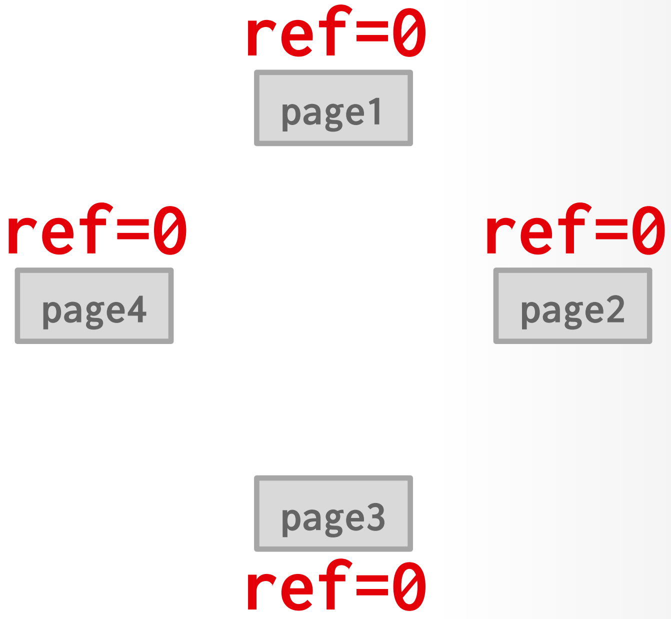
→ Each page has a **reference bit**.

→ 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.



CLOCK (1969)

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

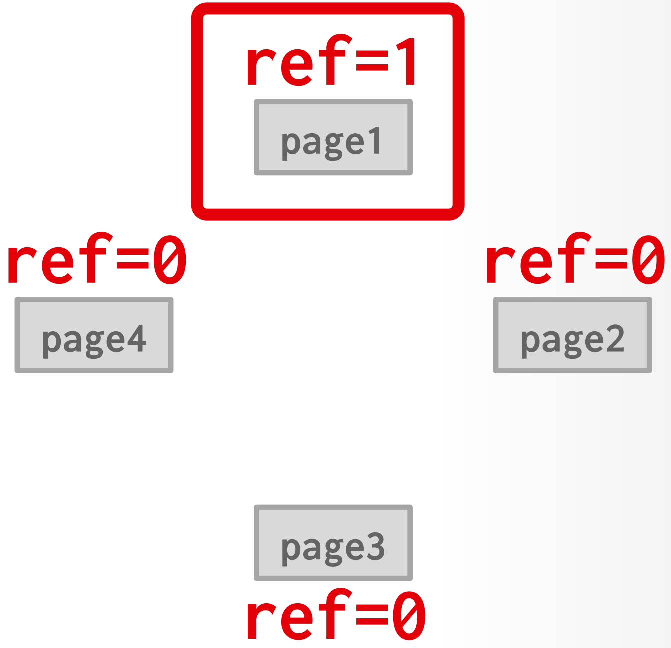
→ Each page has a **reference bit**.

→ 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.

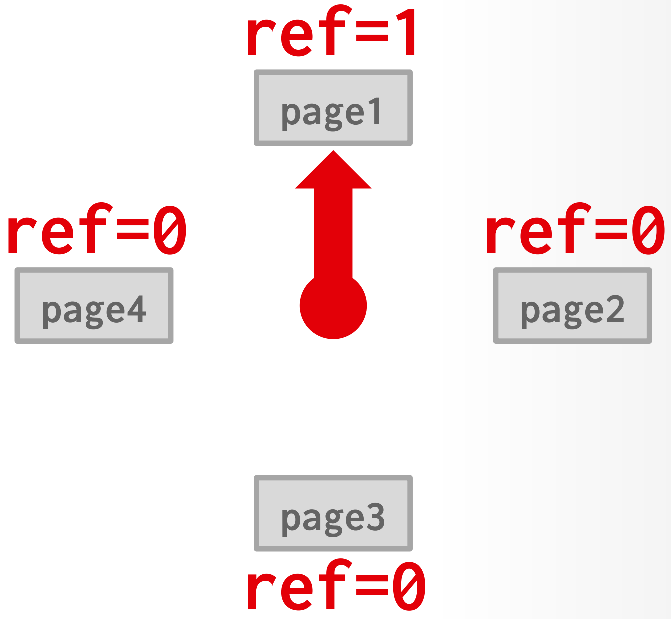
→ If yes, set to zero. If no, then evict.



CLOCK (1969)

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.



CLOCK (1969)

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

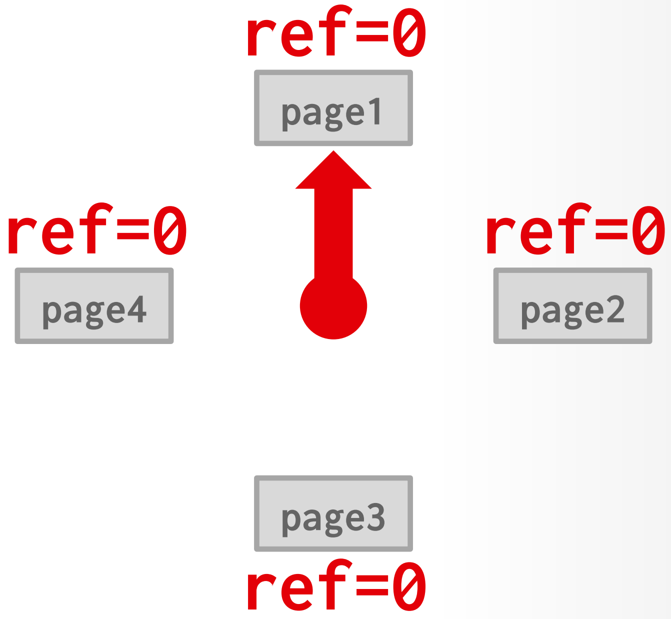
→ Each page has a **reference bit**.

→ 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.



CLOCK (1969)

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

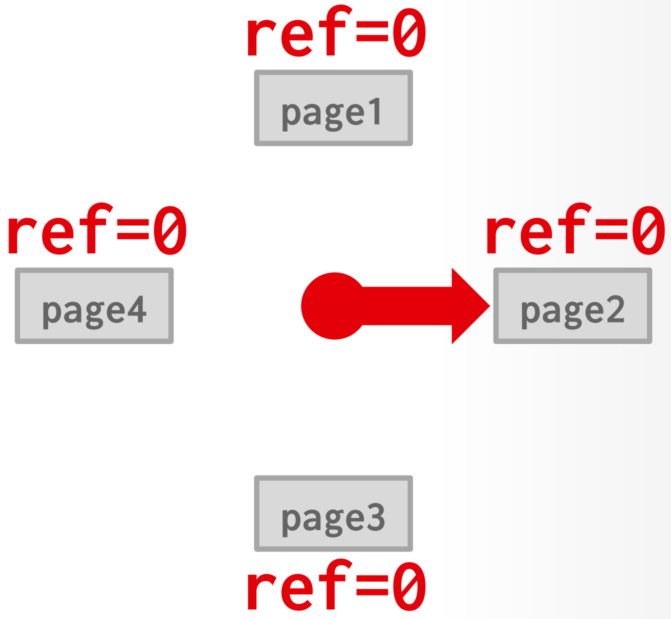
→ Each page has a <u>**reference bit**</u>.

→ 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.



CLOCK (1969)

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

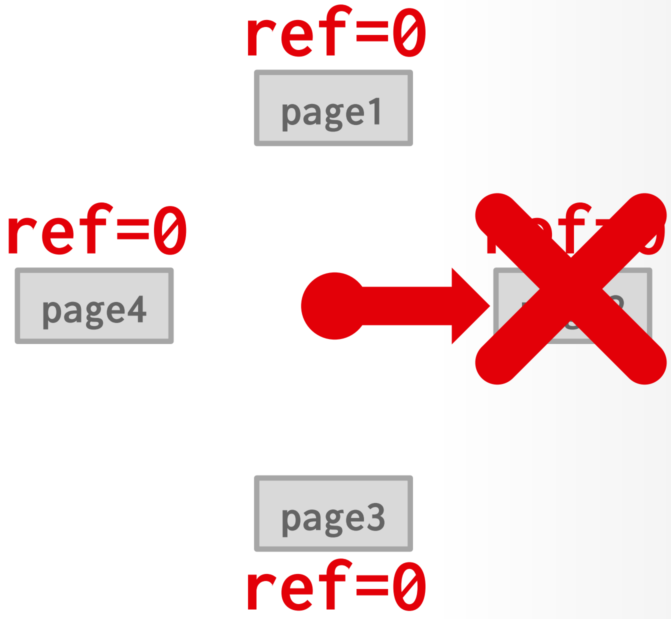
→ Each page has a **reference bit**.

→ 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.



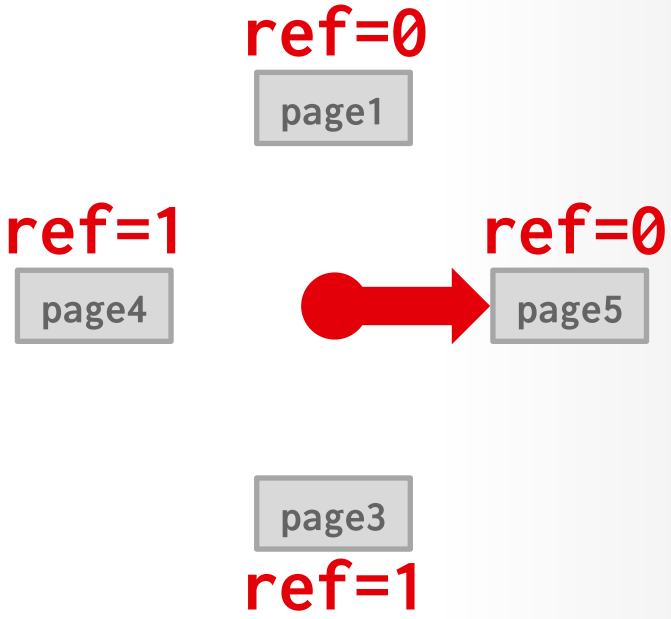
CLOCK (1969)

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.



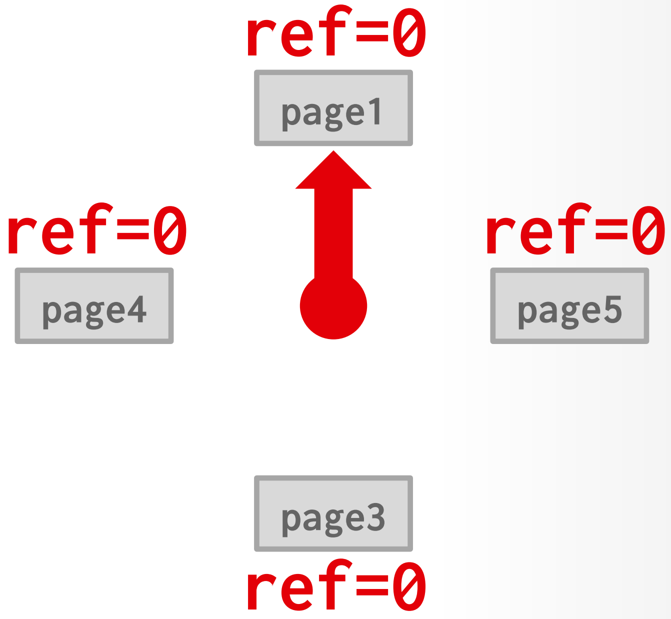
CLOCK (1969)

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:

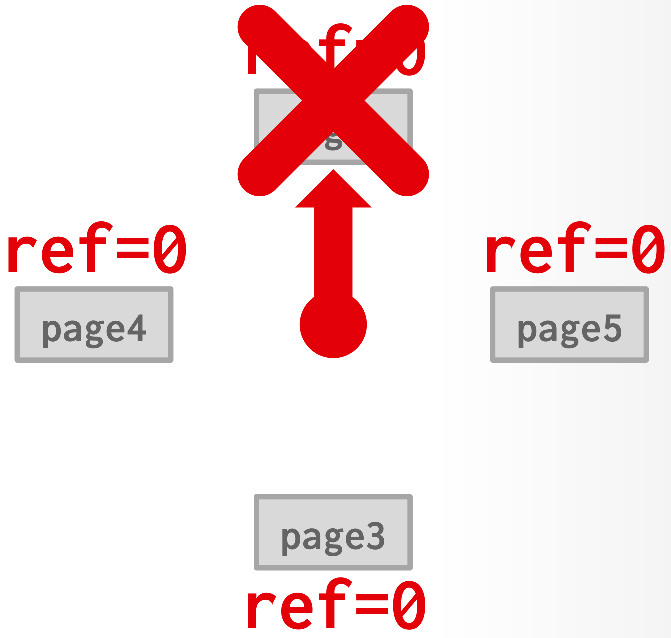
- → As the hand visits each page, check if its bit is set to 1.
- → If yes, set to zero. If no, then evict.



CLOCK (1969)

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

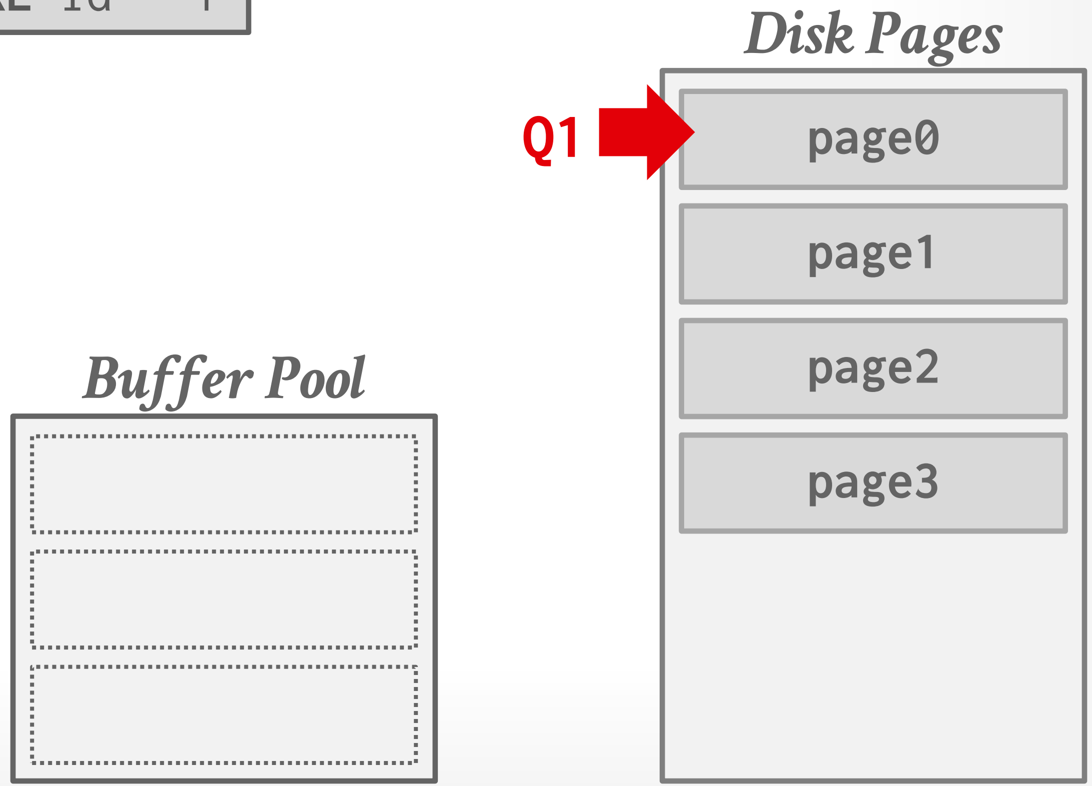
LRU + CLOCK replacement policies are susceptible to **sequential flooding**.

→ A query performs a sequential scan that reads every page in a table one or more times (e.g., nested-loop joins).
→ This pollutes the buffer pool with pages that are read once and then never again.

In OLAP workloads, the ***most recently used*** page is often the best page to evict.

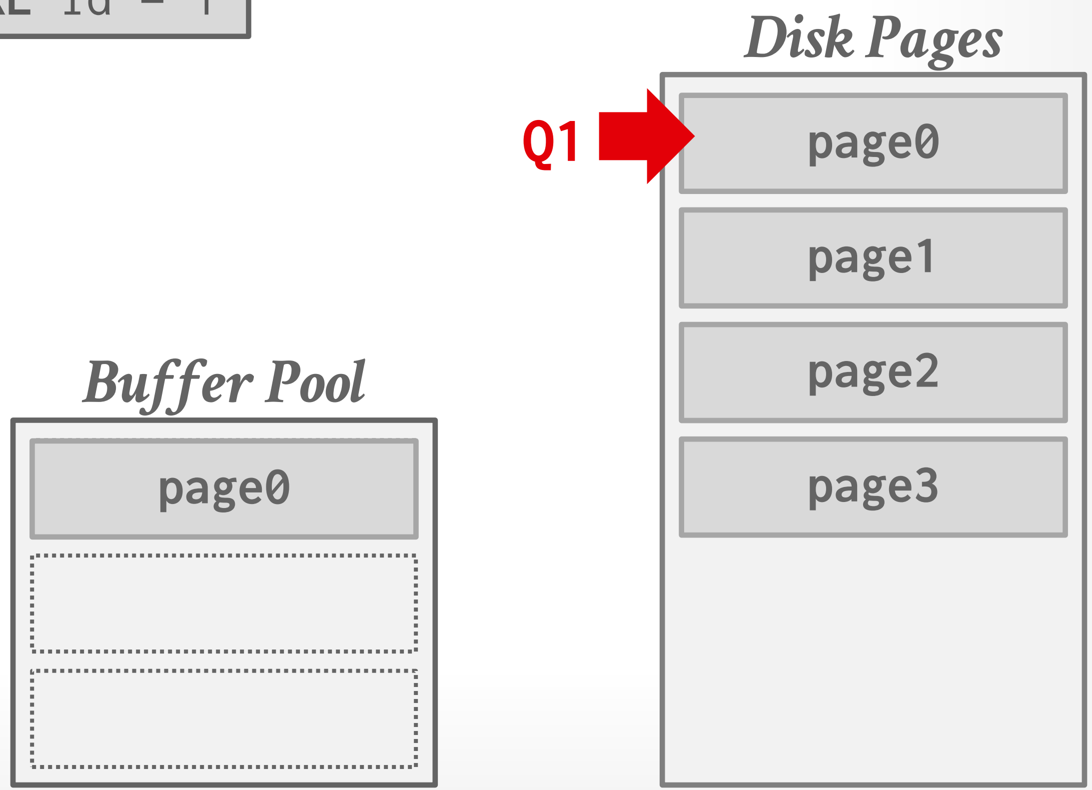
SEQUENTIAL FLOODING

Q1 `SELECT * FROM A WHERE id = 1`



SEQUENTIAL FLOODING

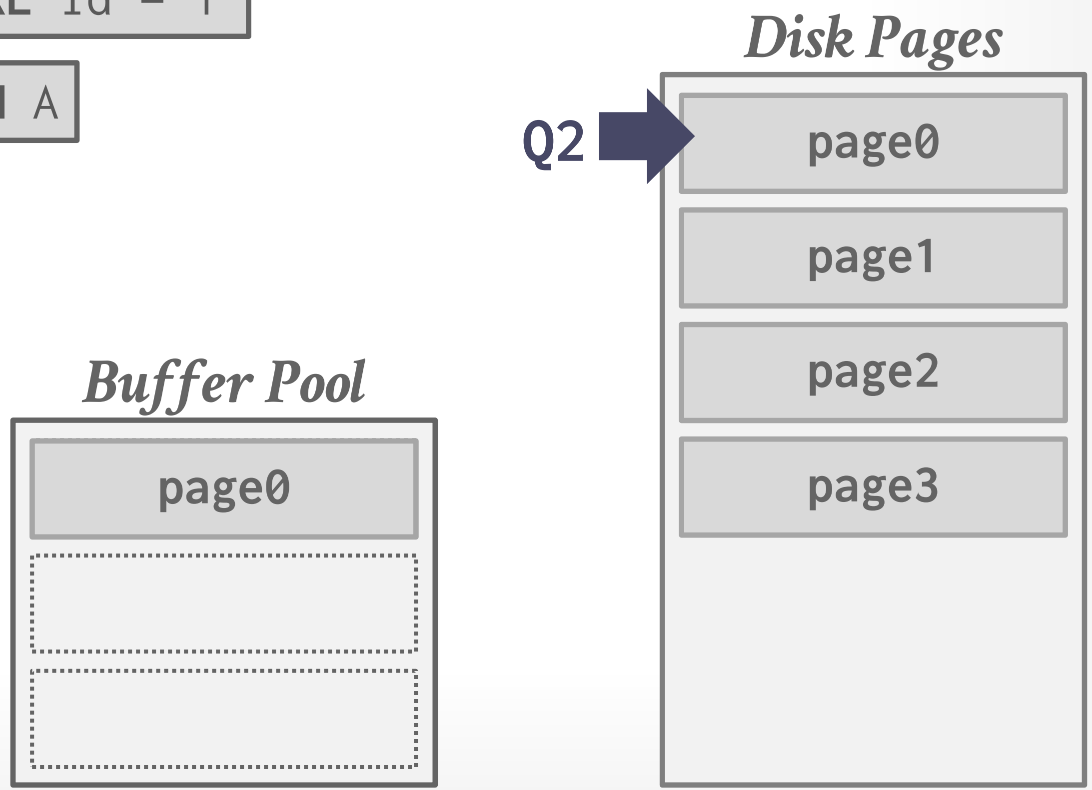
Q1
```
SELECT * FROM A WHERE id = 1
```



SEQUENTIAL FLOODING

Q1 `SELECT * FROM A WHERE id = 1`

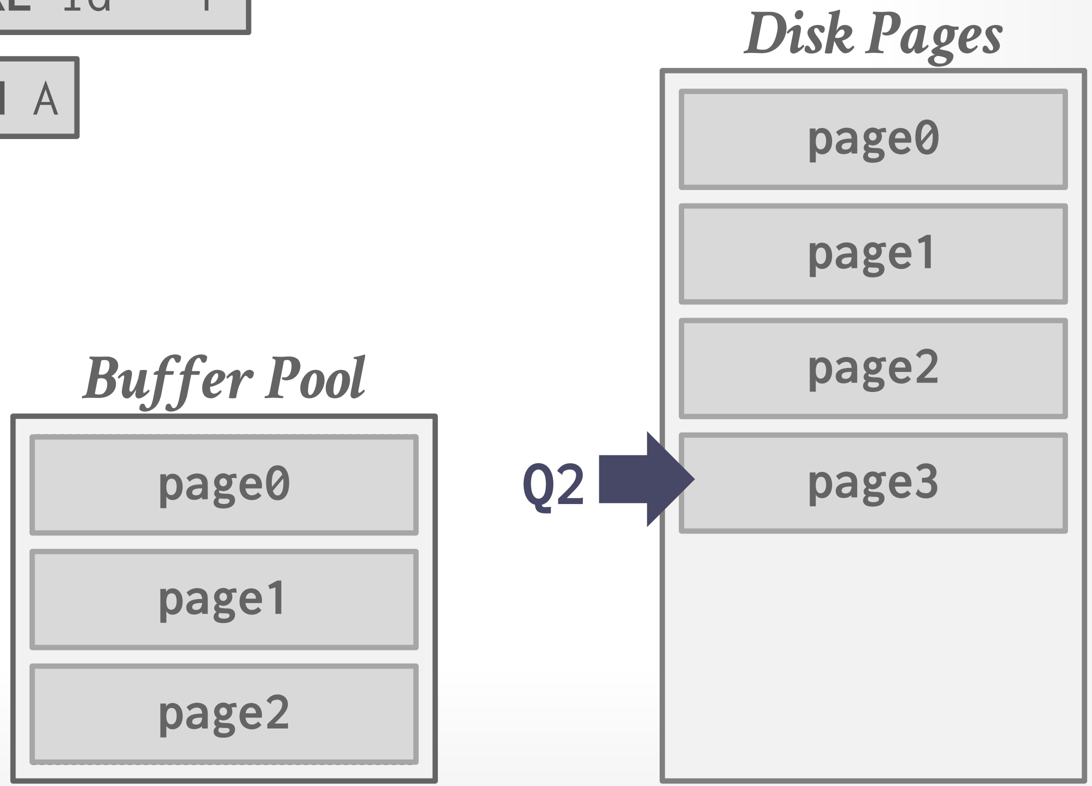
Q2 `SELECT AVG(val) FROM A`



SEQUENTIAL FLOODING

Q1 `SELECT * FROM A WHERE id = 1`

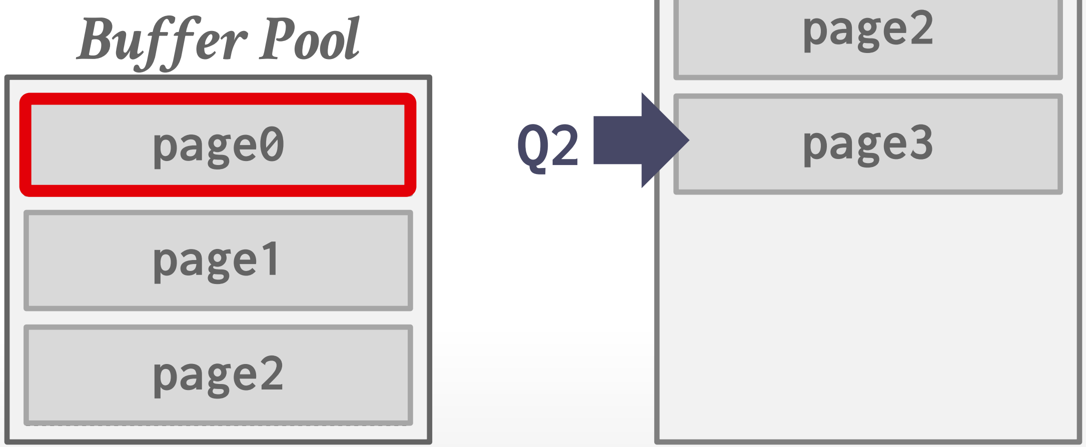
Q2 `SELECT AVG(val) FROM A`



SEQUENTIAL FLOODING

Q1 `SELECT * FROM A WHERE id = 1`

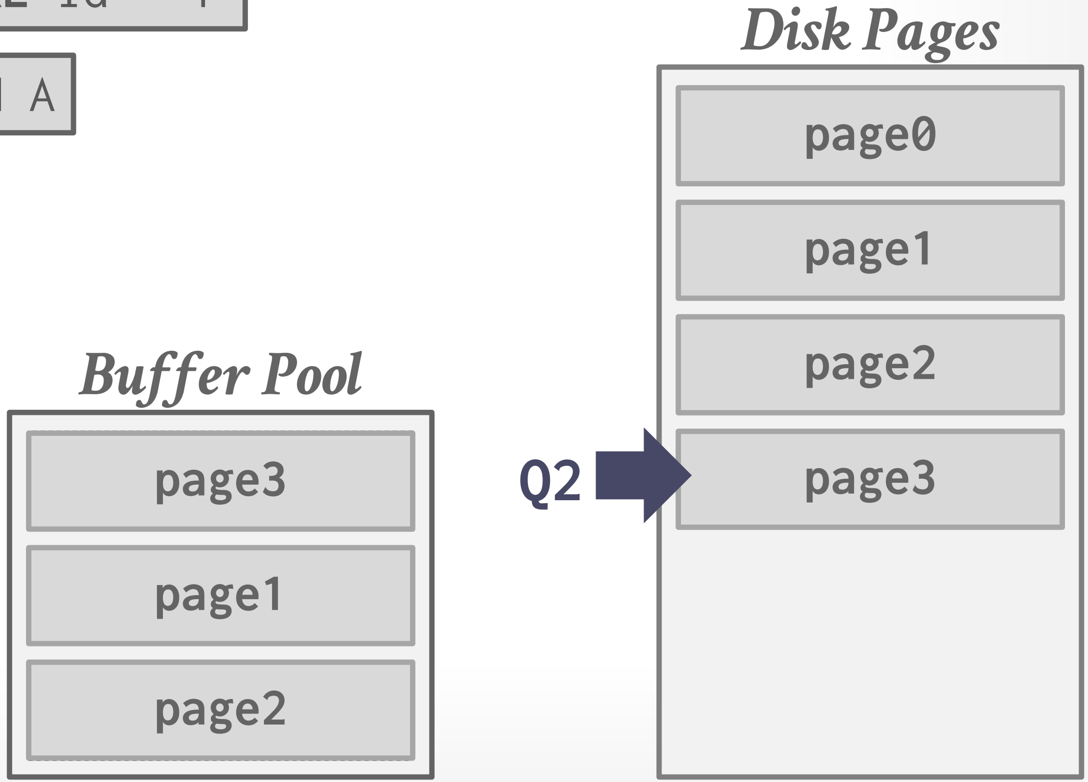
Q2 `SELECT AVG(val) FROM A`



SEQUENTIAL FLOODING

Q1 `SELECT * FROM A WHERE id = 1`

Q2 `SELECT AVG(val) FROM A`

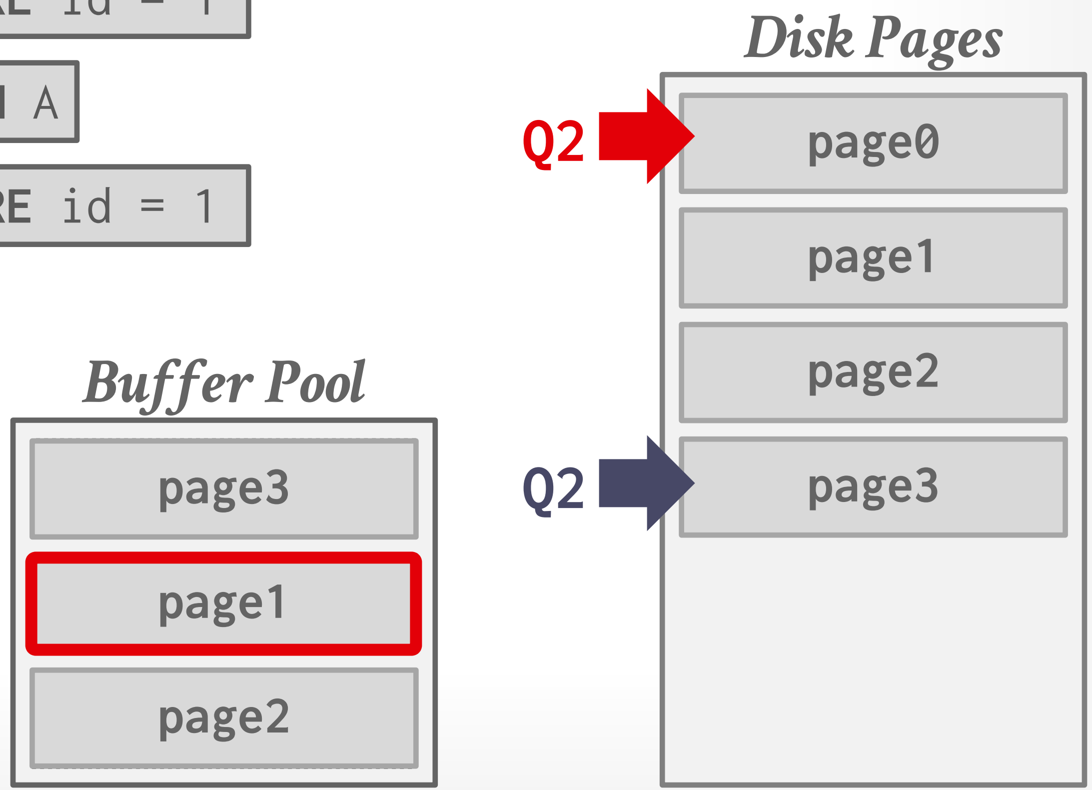


SEQUENTIAL FLOODING

Q1 `SELECT * FROM A WHERE id = 1`

Q2 `SELECT AVG(val) FROM A`

Q3 `SELECT * FROM A WHERE id = 1`



LEAST-FREQUENTLY USED (1971)

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

To identify popular pages, maintain an access count for each page and then evict page with the lowest count.

But LFU introduces more problems:
→ Logarithmic implementation complexity relative to cache size.
→ Ignores time and accumulates stale pages with high frequency counts that may no longer be relevant.

LRU-K (1993)

Track history of last *K* accesses 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 page with the oldest K^{th} access.
→ Balances recency vs. frequency of access.

Maintain in-memory "ghost 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'Neil[1], Patrick E. O'Neil[1], Gerhard Weikum[2]

1 Department of Mathematics and Computer Science, University of Massachussetts at Boston, Harbor Campus, Boston, MA 02125-3393

2 Department of Computer Science, ETH Zurich, CH-8092 Zurich, Switzerland

E-mail: eoneil@cs.umb.edu, poneil@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 pages 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, [COFFDENN], [EFFEHAER]).

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, or to republish, requires a fee and/or specific permission.
SIGMOD /5/93/Washington, DC,USA
© 1993 ACM 0-89791-592-5/93/0005/0297...$1.50

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 logic (for example, [DENNING], [COFFDENN]), and does not always fit well into the database environment, as was noted also in [REITER], [STON], [SACSCH], and [CHOUDEW]. 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 that have 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, free space 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: I1, R1, I2, R2, I3, R3, . . ., alternate references to random index leaf pages and record pages. If we can only afford to buffer 101 pages in memory for this application, the B-tree root node is automatic; we should buffer *all* the B-tree leaf pages, since each of them is referenced with a probability of .005 (once in each 200 general page references), while it is clearly wasteful to displace one of these leaf pages with a data page, since data pages have only .00005 probability of reference (once in each 20,000 general page references). 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

Microsoft® SQL Server®

LRU-K (1993)

Q1
```
SELECT * FROM A WHERE
    id IN (1, 31, 11, 41);
```

Disk Pages

Q1 → page0

page1

page2

page3

page4

page5

Page History

Page	LastAccess #1	LastAccess #2
page0	Time 1	
page1	Time 2	
page2	Time 3	

Buffer Pool

page0

page1

page2

LRU-K (1993)

Q1

```
SELECT * FROM A WHERE
    id IN (1, 31, 11, 41);
```

Disk Pages

Page History

Page	LastAccess #1	LastAccess #2
page0	Time 4	Time 1
page1	Time 2	
page2	Time 3	

Buffer Pool

LRU-K (1993)

Q1

```
SELECT * FROM A WHERE
     id IN (1, 31, 11, 41);
```

Page History

Page	LastAccess #1	LastAccess #2
page0	Time 4	Time 1
page1	Time 2	
page2	Time 3	

Buffer Pool

page0
page1
page2

Q1

Disk Pages

page0
page1
page2
page3
page4
page5

LRU-K (1993)

Q1
```
SELECT * FROM A WHERE
     id IN (1, 31, 11, 41);
```

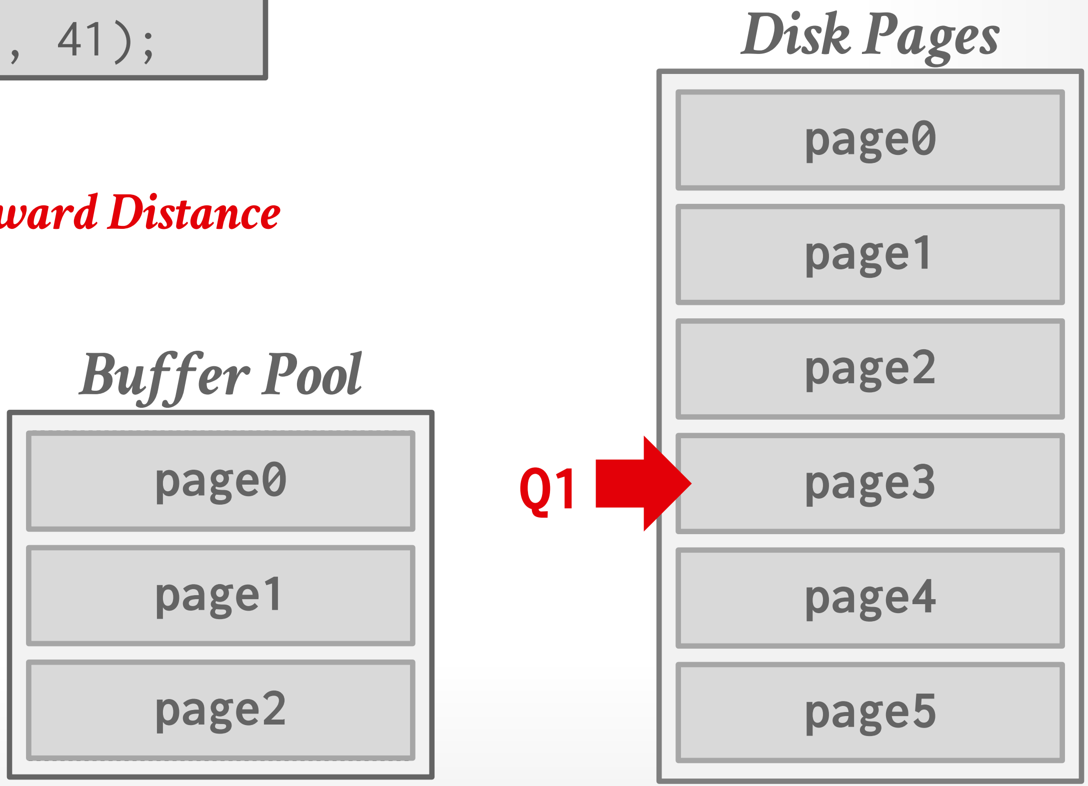
Eviction Policy:

CurrentTime – k^{th} LastAccess = Backward Distance

Use oldest access time to break ties.

Page History

Page	LastAccess #1	LastAccess #2
page0	Time 4	Time 1
page1	Time 2	
page2	Time 3	



LRU-K (1993)

Q1
```
SELECT * FROM A WHERE
     id IN (1, 31, 11, 41);
```

Eviction Policy:

CurrentTime – k^{th} LastAccess = Backward Distance

Use oldest access time to break ties.

Page History

Page	LastAccess #1	LastAccess #2
page0	Time 4	Time 1
page1	Time 2	
page2	Time 3	

Time=5

4

∞

∞

Buffer Pool

page0

page1

page2

Disk Pages

page0

page1

page2

Q1 → page3

page4

page5

LRU-K (1993)

Q1

```
SELECT * FROM A WHERE
    id IN (1, 31, 11, 41);
```

Eviction Policy:

CurrentTime – k^{th} LastAccess = Backward Distance

Use oldest access time to break ties.

Page History

Page	LastAccess #1	LastAccess #2	*Time=5*
page0	Time 4	Time 1	4
page3	Time 5		∞
page2	Time 3		∞

Buffer Pool

- page0
- page3
- page2

Q1

Disk Pages

- page0
- page1
- page2
- page3
- page4
- page5

LRU-K (1993)

Q1

```
SELECT * FROM A WHERE
     id IN (1, 31, 11, 41);
```

Eviction Policy:

CurrentTime – k^th LastAccess = Backward Distance

Use oldest access time to break ties.

Page History

Page	LastAccess #1	LastAccess #2
page0	Time 4	Time 1
page3	Time 5	
page2	Time 3	

Buffer Pool

page0

page3

page2

Disk Pages

page0

Q1 → page1

page2

page3

page4

page5

LRU-K (1993)

Q1
```
SELECT * FROM A WHERE
    id IN (1, 31, 11, 41);
```

Eviction Policy:

CurrentTime – k^{th} LastAccess = Backward Distance

Use oldest access time to break ties.

Page History

Page	LastAccess #1	LastAccess #2	Time=6
page0	Time 4	Time 1	5
page3	Time 5		∞
page2	Time 3		∞

Buffer Pool

- page0
- page3
- page2

Disk Pages

Q1 → page1

- page0
- page1
- page2
- page3
- page4
- page5

LRU-K (1993)

Q1
```
SELECT * FROM A WHERE
    id IN (1, 31, 11, 41);
```

Eviction Policy:

CurrentTime – k^{th} LastAccess = Backward Distance

Use oldest access time to break ties.

Page History

Page	LastAccess #1	LastAccess #2	*Time=6*
page0	Time 4	Time 1	5
page3	Time 5		∞
page1	Time 6	Time 2	∞

Buffer Pool

page0

page3

page1

Disk Pages

page0

Q1 → page1

page2

page3

page4

page5

LRU-K (1993)

Q1
```
SELECT * FROM A WHERE
      id IN (1, 31, 11, 41);
```

Eviction Policy:

CurrentTime – k^{th} LastAccess = Backward Distance

Use oldest access time to break ties.

Page History

Page	LastAccess #1	LastAccess #2
page0	Time 4	Time 1
page3	Time 5	
page1	Time 6	Time 2

Time=7

6

∞

5

Buffer Pool

page0

page3

page1

Q1

Disk Pages

page0

page1

page2

page3

page4

page5

LRU-K (1993)

Q1
```
SELECT * FROM A WHERE
      id IN (1, 31, 11, 41);
```

Eviction Policy:

CurrentTime – k^{th} LastAccess = Backward Distance

Use oldest access time to break ties.

Page History

Page	LastAccess #1	LastAccess #2	*Time=7*
page0	Time 4	Time 1	6
page4	Time 7		∞
page1	Time 6	Time 2	5

Buffer Pool

page0

page4

page1

Disk Pages

page0

page1

page2

page3

Q1 → page4

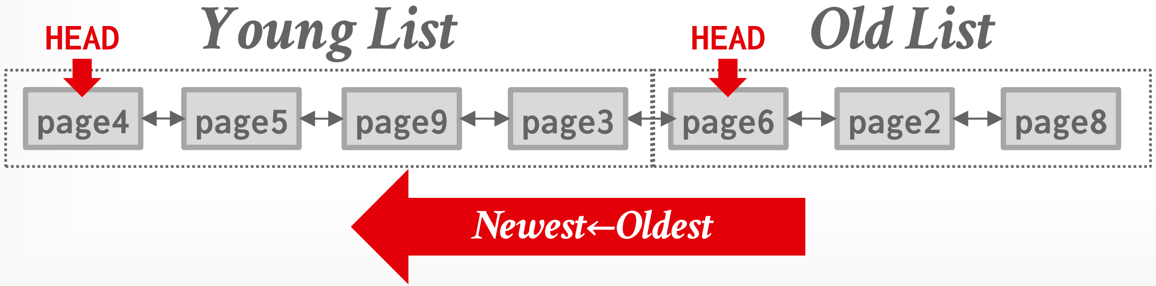
page5

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.

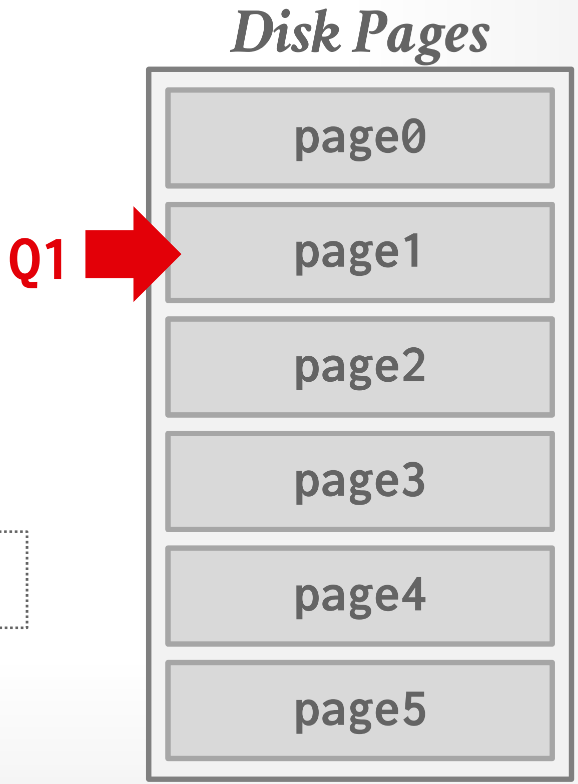


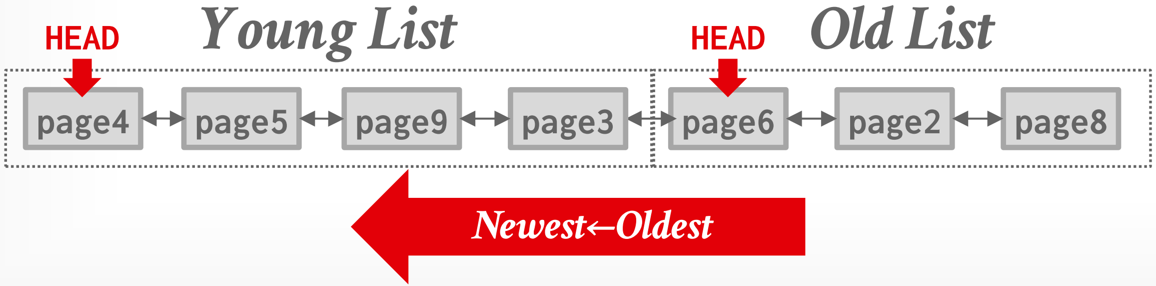
MYSQL APPROXIMATE LRU-K

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.



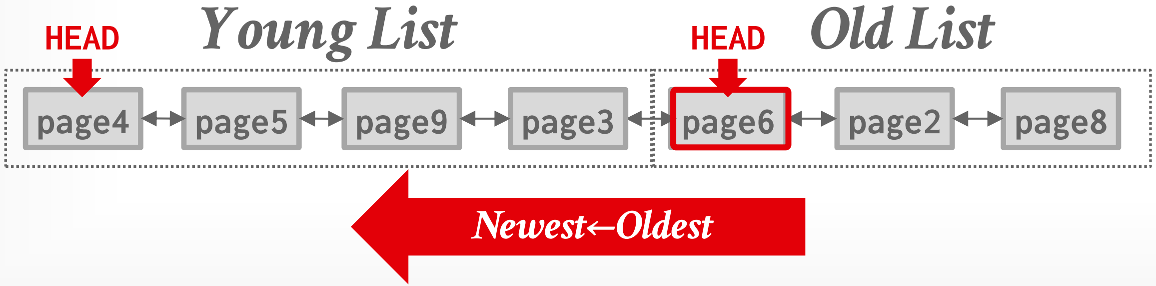


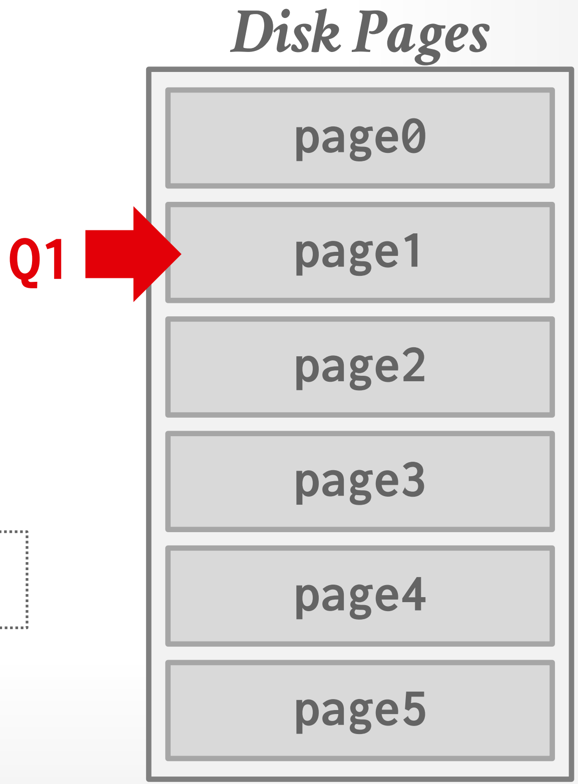
MYSQL APPROXIMATE LRU-K

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.



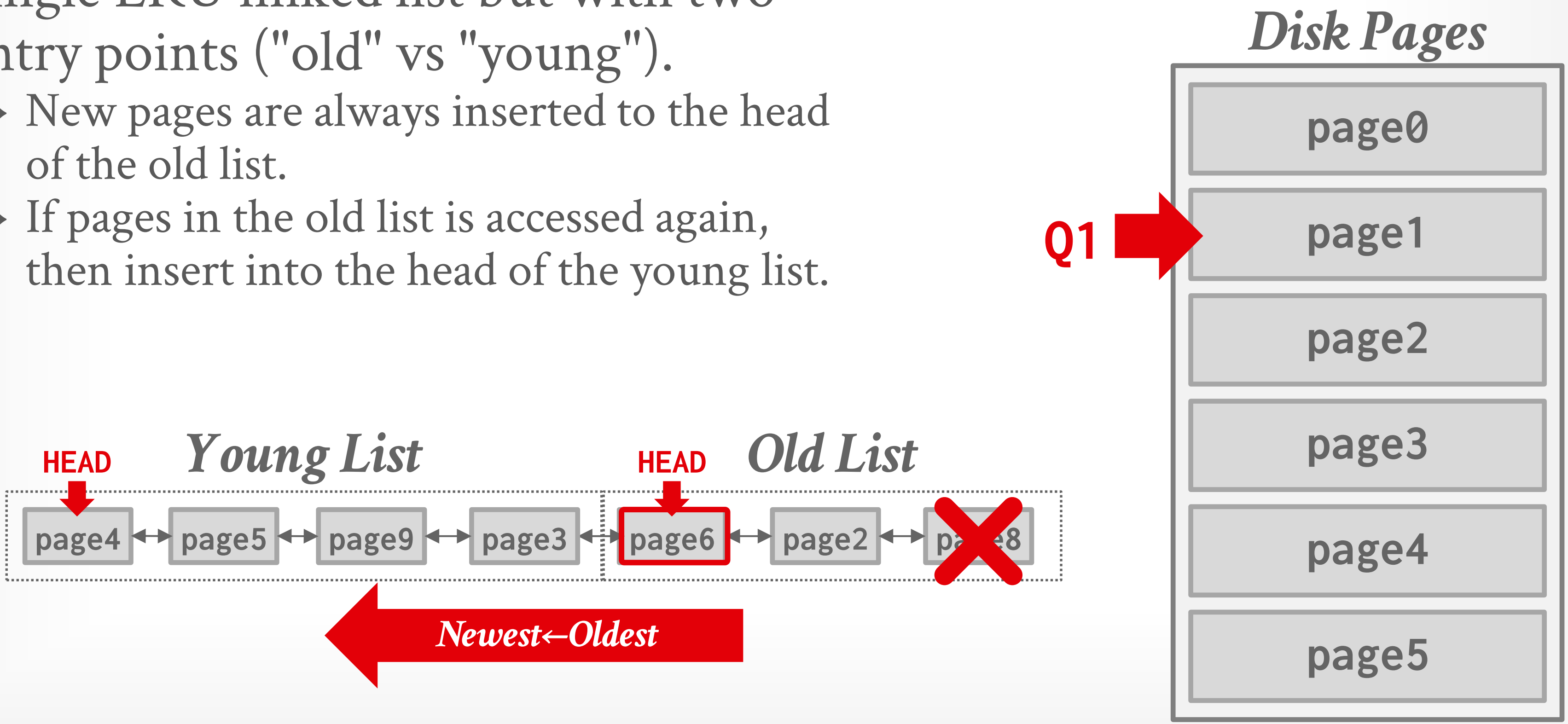


MYSQL APPROXIMATE LRU-K

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

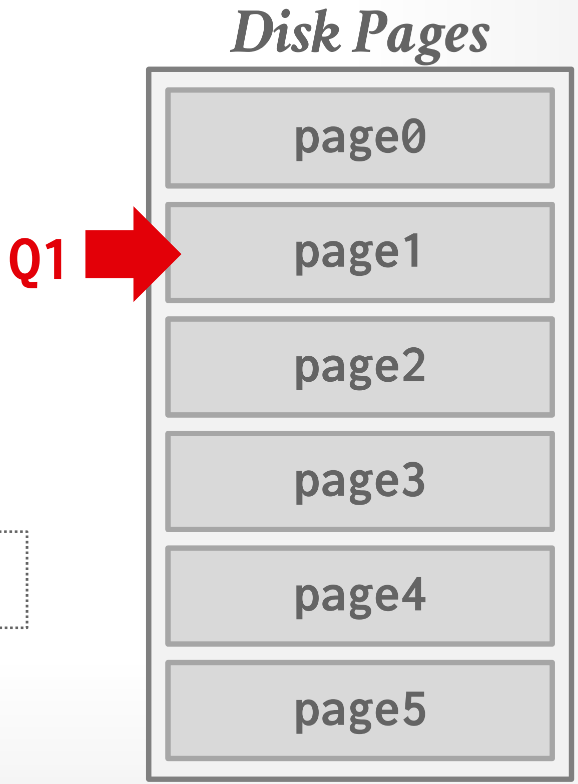
→ New pages are always inserted to the head of the old list.

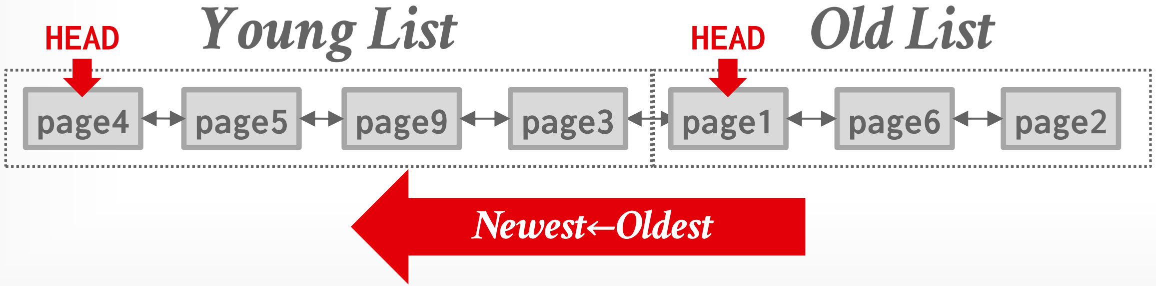
→ If pages in the old list is accessed again, then insert into the head of the young list.



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.



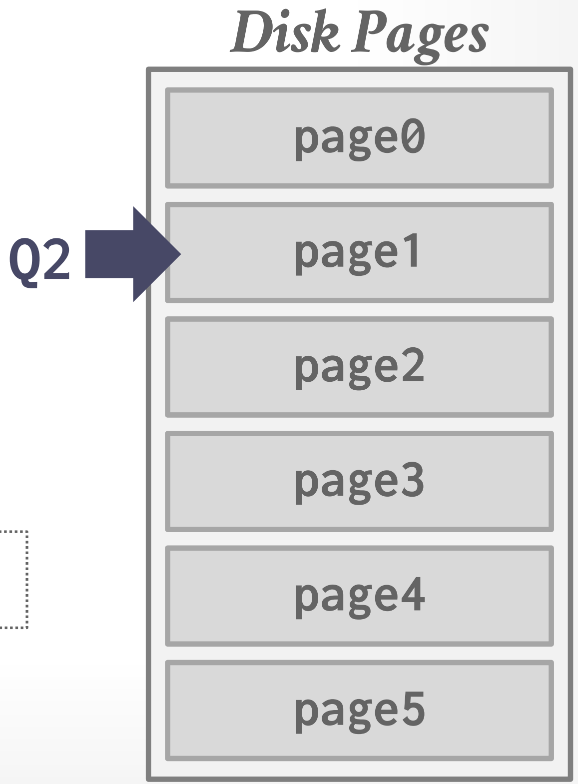


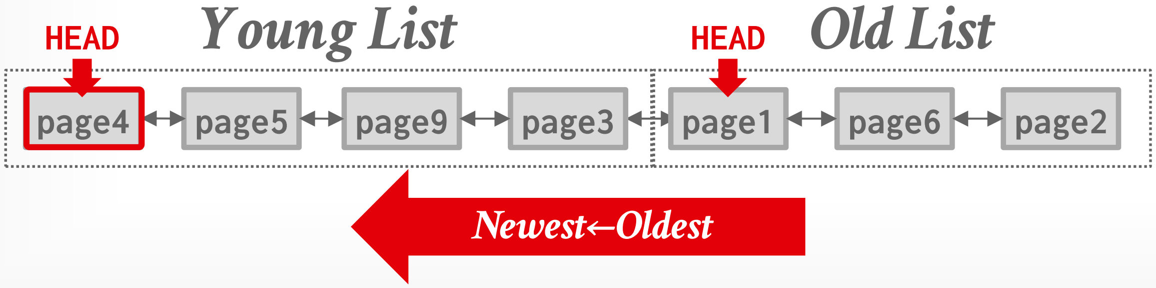
MYSQL APPROXIMATE LRU-K

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.



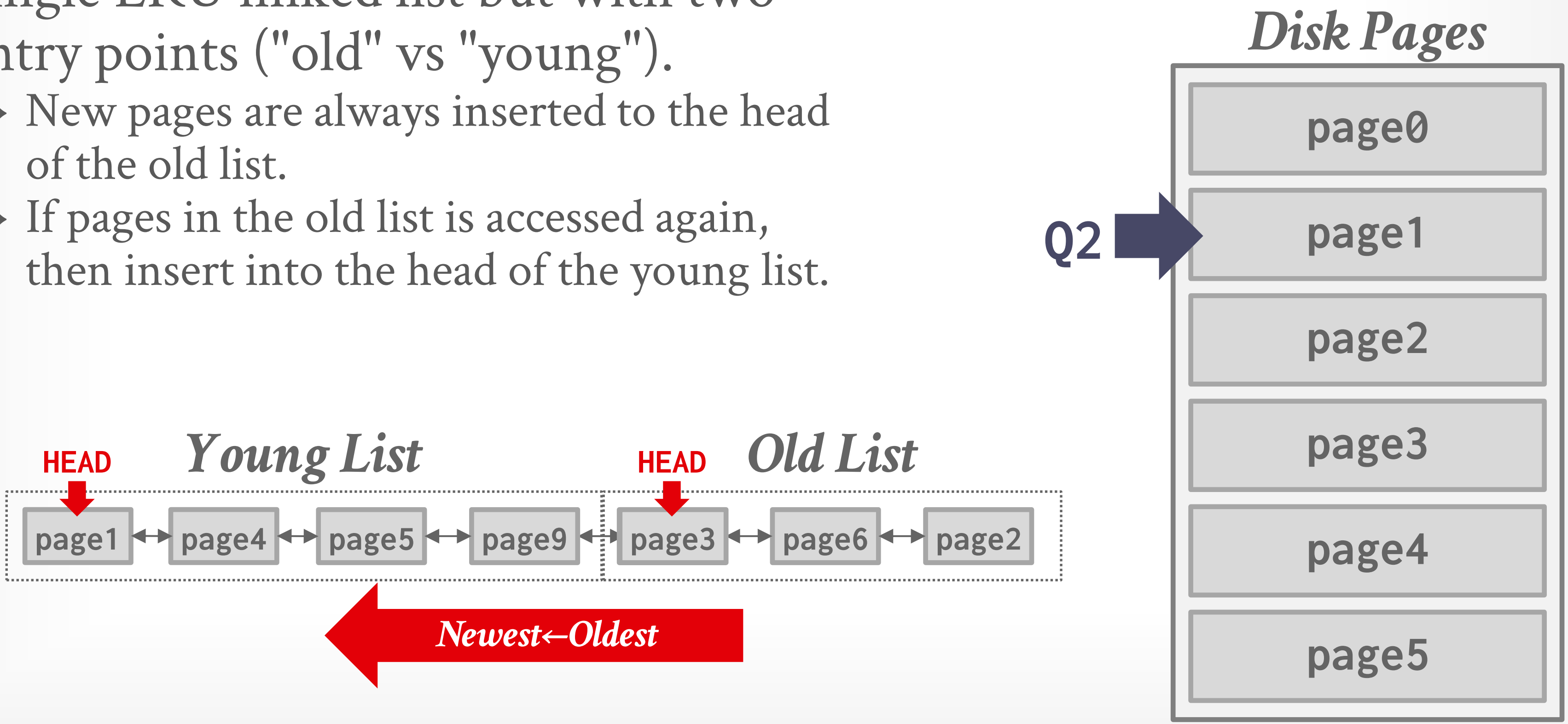


MYSQL APPROXIMATE LRU-K

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.



ARC: ADAPTIVE REPLACEMENT CACHE (2003)

Adaptive replacement policy algorithm developed by IBM Research.
→ Only implemented in IBM DB2, PostgreSQL, and ZFS.
→ Rewritten in PostgreSQL to avoid IBM's patent.

Support both recency (LRU) and frequency (LFU) by maintaining two lists and then adjusts the size of them based on workload access patterns.

Maintain ghost lists to remember recent evictions and adapt quickly.

ARC: ADAPTIVE REPLACEMENT CACHE (2003)

Recent List (T1):
→ Holds pages that have been accessed once recently.

Recent Ghost List (B1):
→ History of pages recently evicted from **T1** (i.e., recency misses).

Frequent List (T2):
→ Holds pages that have been accessed at least twice.

Frequent Ghost List (B2):
→ History of pages recently evicted from **T2** (i.e., frequency misses).

Target Size Parameter (p):
→ Adaptively adjusts how much to favor recency (**T1**) vs. frequency (**T2**).

ARC: LOOKUP PROTOCOL

A page can only exist in one list at a time.

The DBMS checks both lists upon look-up to determine whether it is available.

Page Found in T1:

→ Page is promoted from **T1** → **T2** (i.e., becomes "frequent").

Page Found in T2:

→ Page remains in **T2**, reinforcing its "frequent" status.

ARC: LOOKUP PROTOCOL

Cache Miss, Page Found in B1 (Ghost of T1):

→ Increase target size **p** (favor more recency pages).
→ Move page into **T2** (since it's now accessed again).

Cache Miss, Page Found in B2 (Ghost of T2):

→ Decrease target size **p** (favor more frequency pages).
→ Move page into **T2**.

Cache Miss, Page Not in Cache or Ghost Lists:

→ If **T1** + **B1** is full, evict from **B1** or **T1**.
→ If **T2** + **B2** is full, evict from **B2** or **T2**.
→ Insert new page into **T1**.

ARC: LOOKUP PROTOCOL

Cache Miss, Page Found in B1 (Ghost of T1):

→ Increase target size **p** (favor more recency pages).
→ Move page into **T2** (since it's now accessed again).

Cache Miss, Page Found in B2 (Ghost of T2):

→ Decrease target size **p** (favor more frequency pages).
→ Move page into **T2**.

Cache Miss, Page Not in Cache or Ghost Lists:

→ If **T1** + **B1** is full, evict from **B1** or **T1**.
→ If **T2** + **B2** is full, evict from **B2** or **T2**.
→ Insert new page into **T1**.

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.

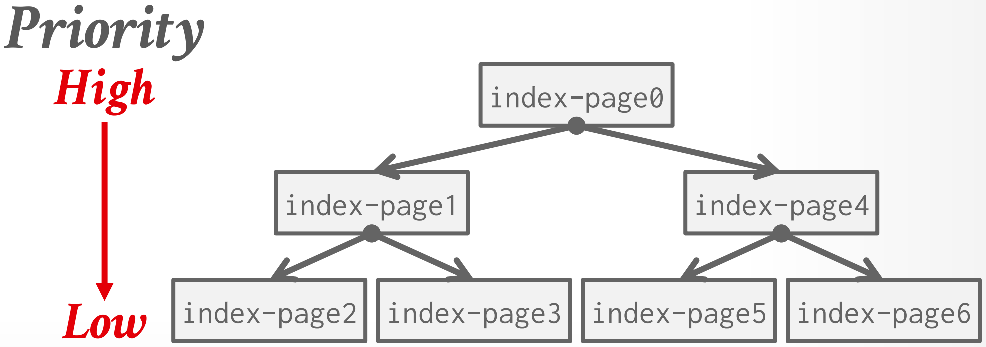
→ Keep track of the pages that a query has accessed.

Example: Postgres assigns a limited number of buffer 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.

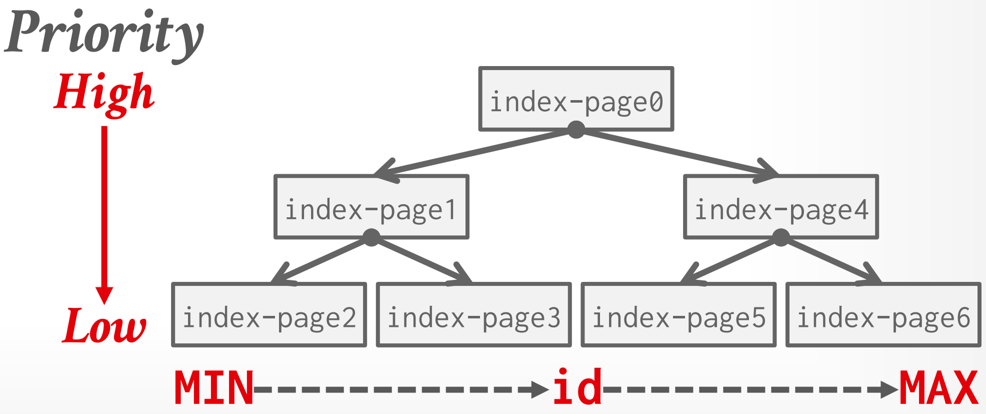


BETTER POLICIES: PRIORITY HINTS

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++)`

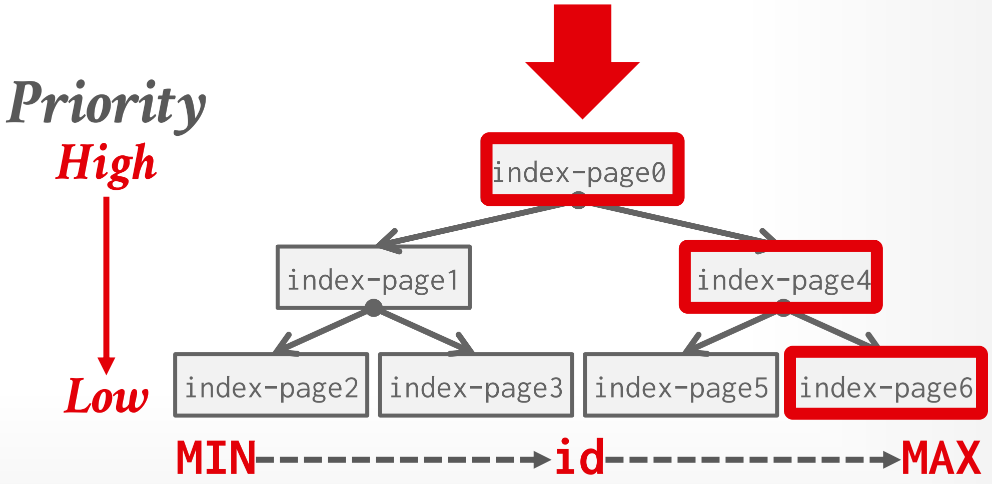


BETTER POLICIES: PRIORITY HINTS

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++)`



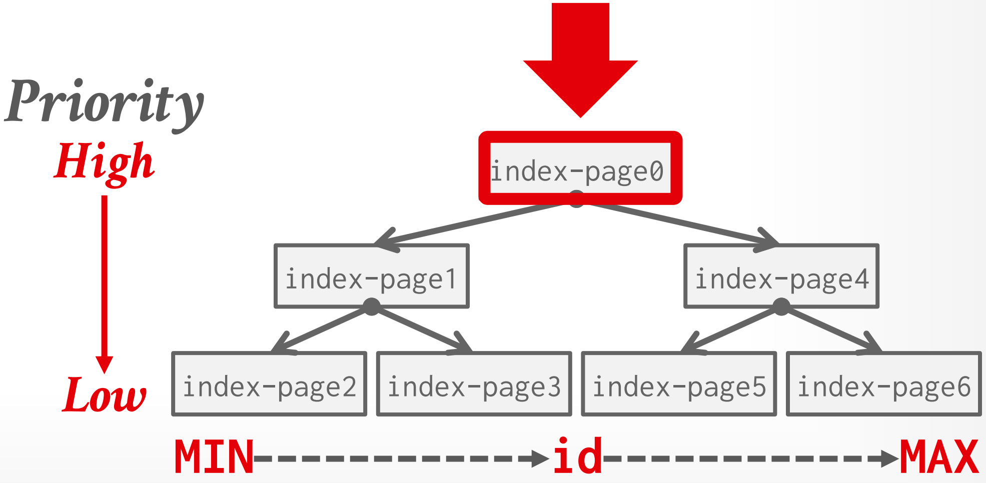
BETTER POLICIES: PRIORITY HINTS

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 dirty writing pages that will not be read again in the future.

BACKGROUND WRITING

The DBMS periodically walks through the page table and preemptively write dirty pages to disk.

→ Also called page cleaning or "buffer flushing".

When a dirty page is safely flushed, the DBMS can either evict the page or just reset its dirty flag.

Need to be careful the system does not write dirty pages before their log records are written…

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 maintain 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-based SLAs

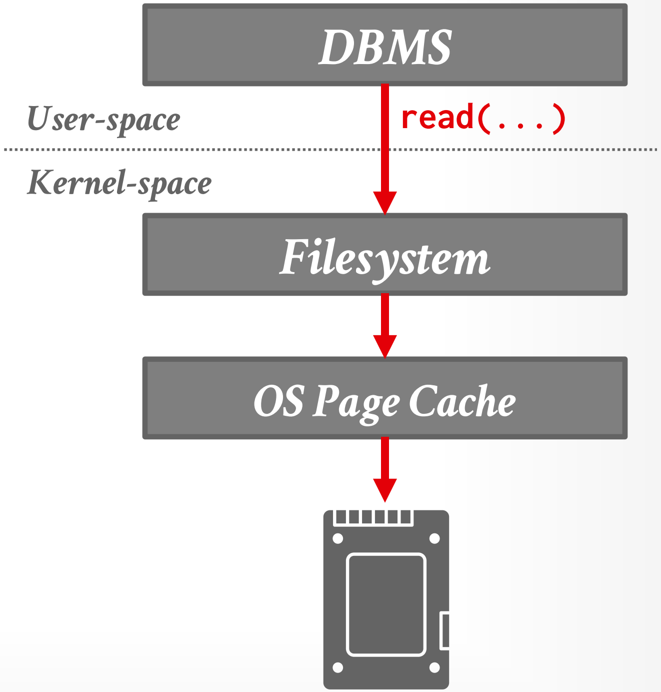
The OS doesn't know these things and is going to get in the way of our beautiful DBMS…

OS PAGE CACHE

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.

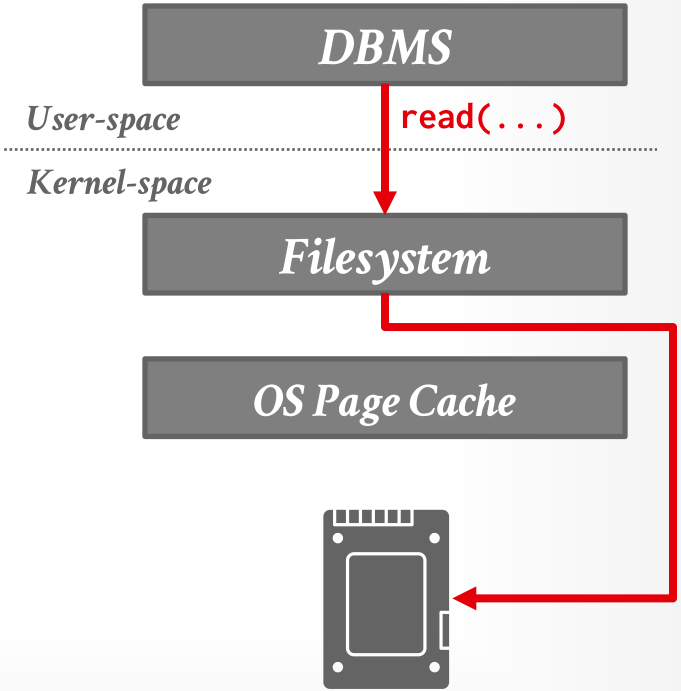
→ Redundant copies of pages.
→ Different eviction policies.
→ Loss of control over file I/O.



OS PAGE CACHE

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.

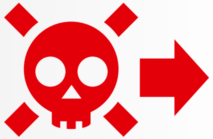


FSYNC PROBLEMS

If the DBMS calls **`fwrite`**, what happens?

If the DBMS calls **`fsync`**, what happens?

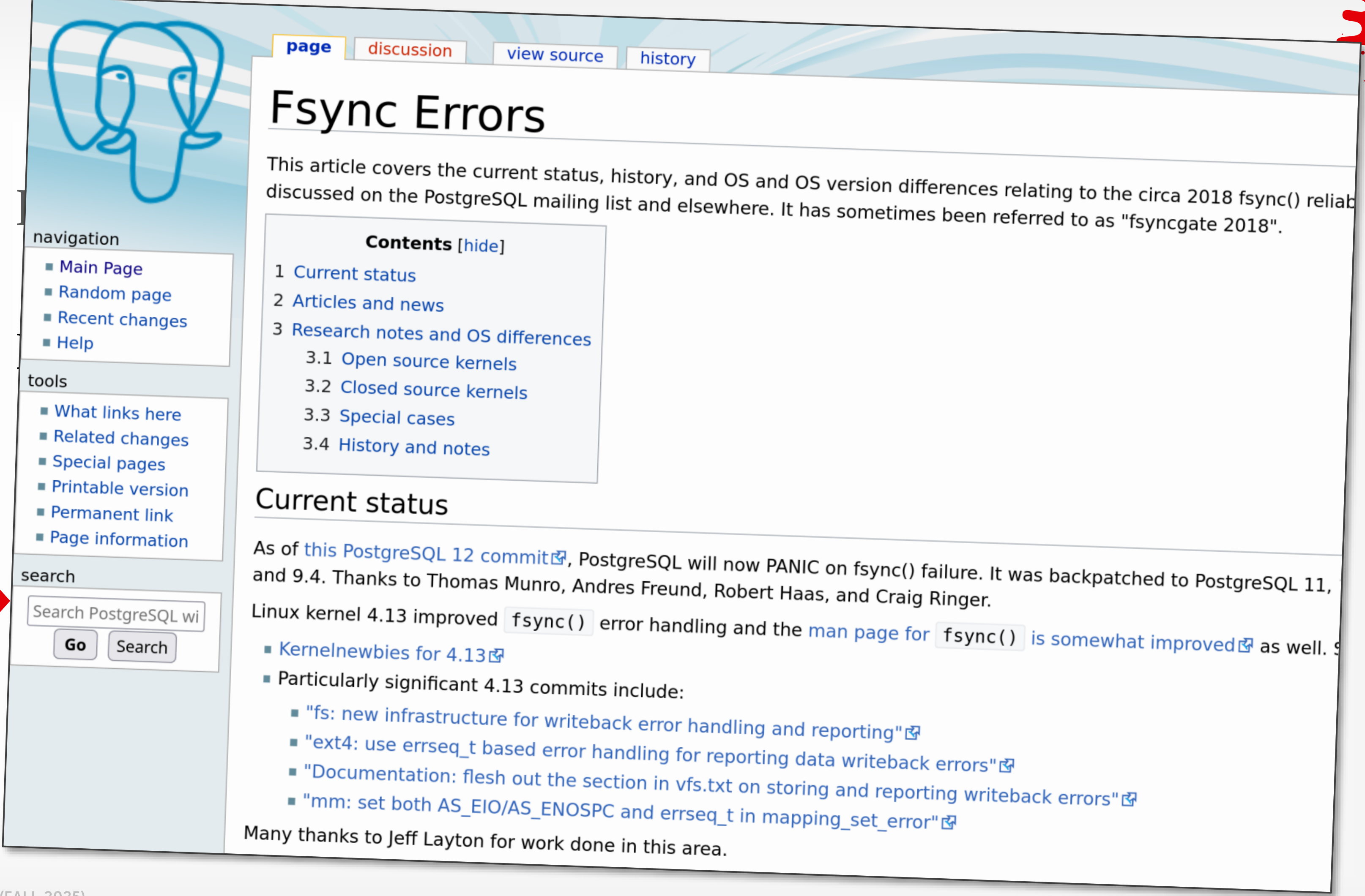
If **`fsync`** fails (EIO), what happens?



Don't Do This!

→ Linux marks the dirty pages as clean.

→ If the DBMS calls **`fsync`** again, then Linux tells you that the flush was successful. Since the DBMS thought the OS was its friend, it assumed the write was successful…



page | discussion | view source | history

Fsync Errors

This article covers the current status, history, and OS and OS version differences relating to the circa 2018 fsync() reliab
discussed on the PostgreSQL mailing list and elsewhere. It has sometimes been referred to as "fsyncgate 2018".

Contents [hide]

Current status

As of this PostgreSQL 12 commit, PostgreSQL will now PANIC on fsync() failure. It was backpatched to PostgreSQL 11,
and 9.4. Thanks to Thomas Munro, Andres Freund, Robert Haas, and Craig Ringer.

Linux kernel 4.13 improved `fsync()` error handling and the man page for `fsync()` is somewhat improved as well.

- Kernelnewbies for 4.13
- Particularly significant 4.13 commits include:
  - "fs: new infrastructure for writeback error handling and reporting"
  - "ext4: use errseq_t based error handling for reporting data writeback errors"
  - "Documentation: flesh out the section in vfs.txt on storing and reporting writeback errors"
  - "mm: set both AS_EIO/AS_ENOSPC and errseq_t in mapping_set_error"

Many thanks to Jeff Layton for work done in this area.

navigation
- Main Page
- Random page
- Recent changes
- Help

tools
- What links here
- Related changes
- Special pages
- Printable version
- Permanent link
- Page information

search
Search PostgreSQL wi
Go | Search

Don't Do This!

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

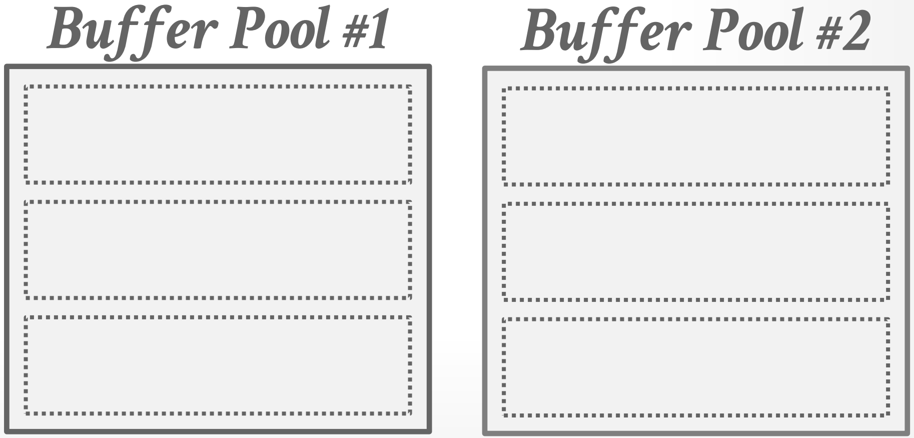
Partitioning memory across multiple pools helps reduce latch contention and improve locality.
→ Avoids contention on LRU tracking meta-data.

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.

Q1 `GET RECORD #123`



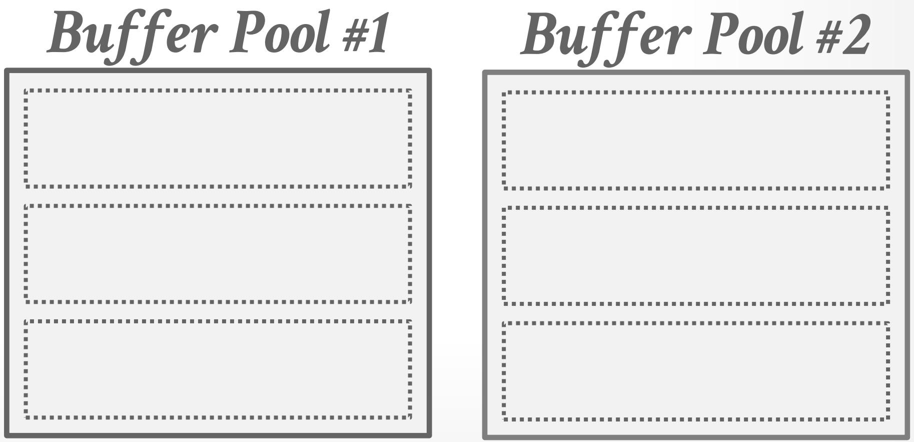
MULTIPLE BUFFER POOLS

Approach #1: Object Id

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

Q1 `GET RECORD #123`

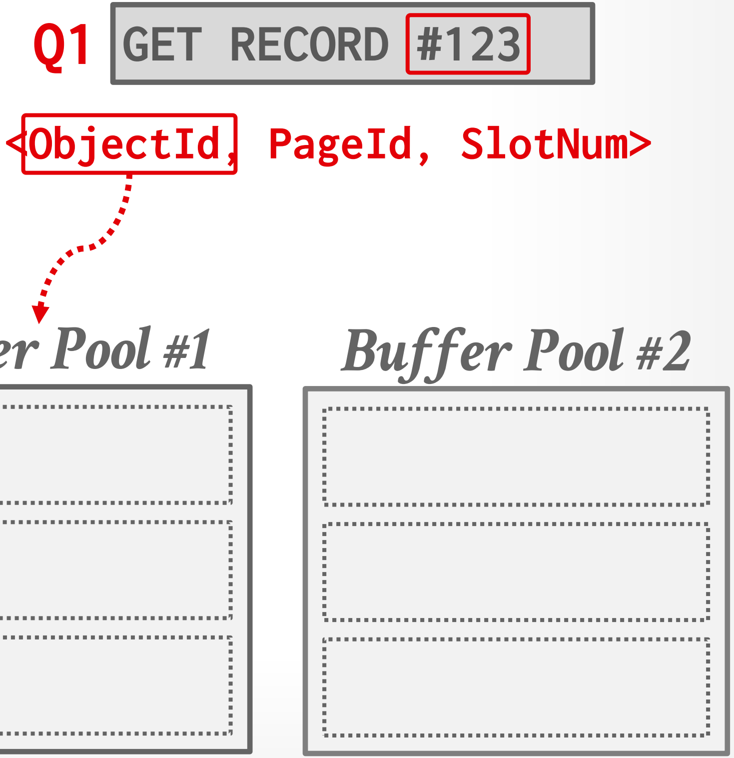
`<ObjectId, PageId, SlotNum>`



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.



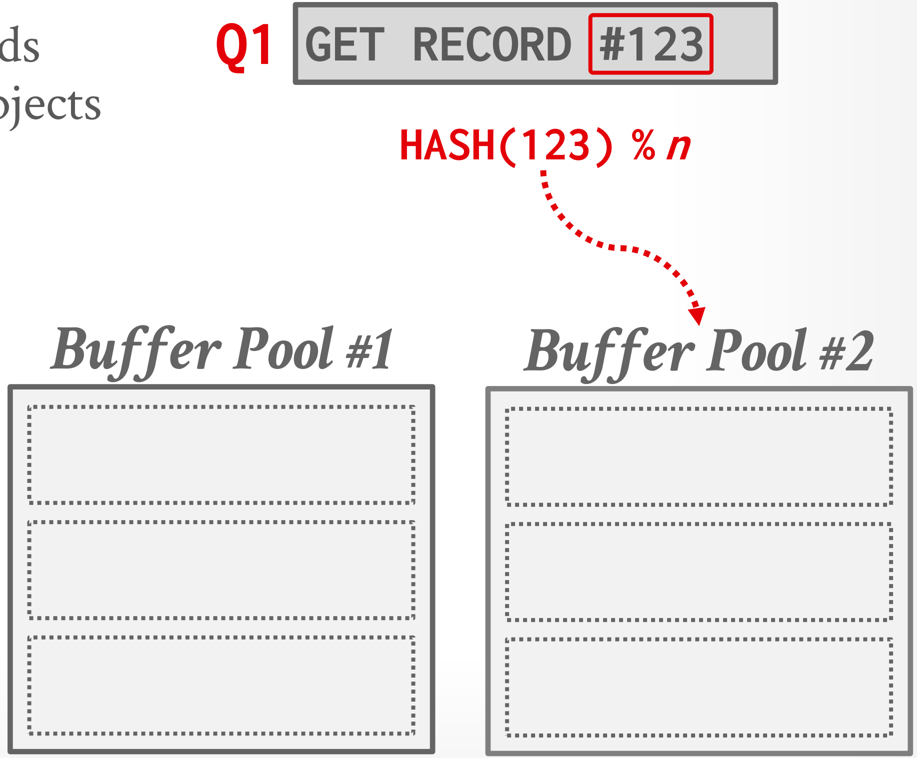
MULTIPLE BUFFER POOLS

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.

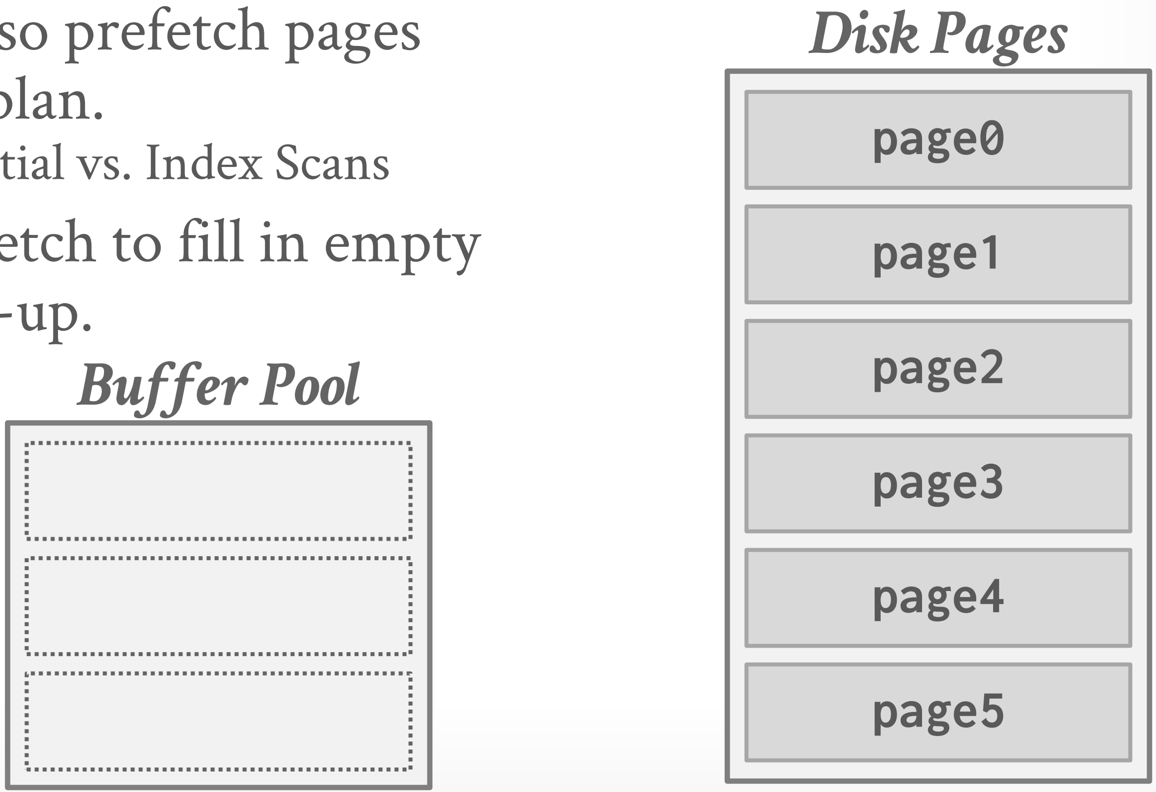


PRE-FETCHING

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.

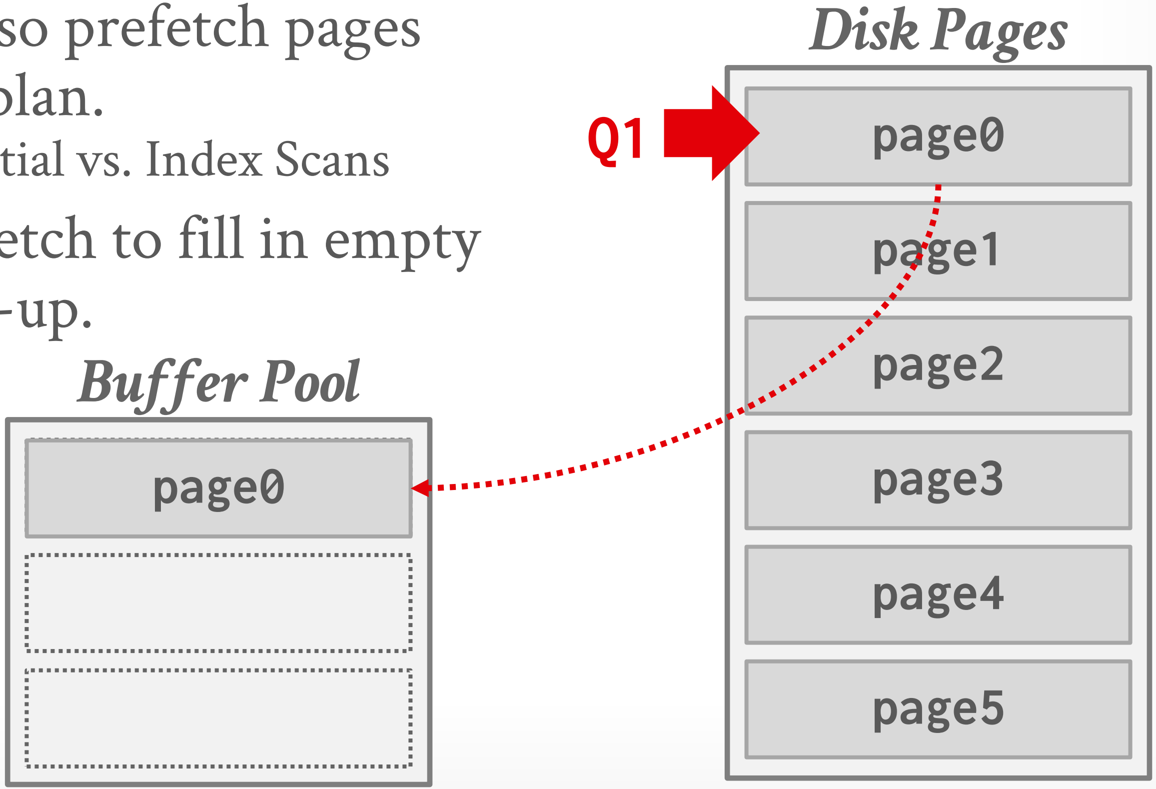


PRE-FETCHING

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

→ Examples: Sequential vs. Index Scans

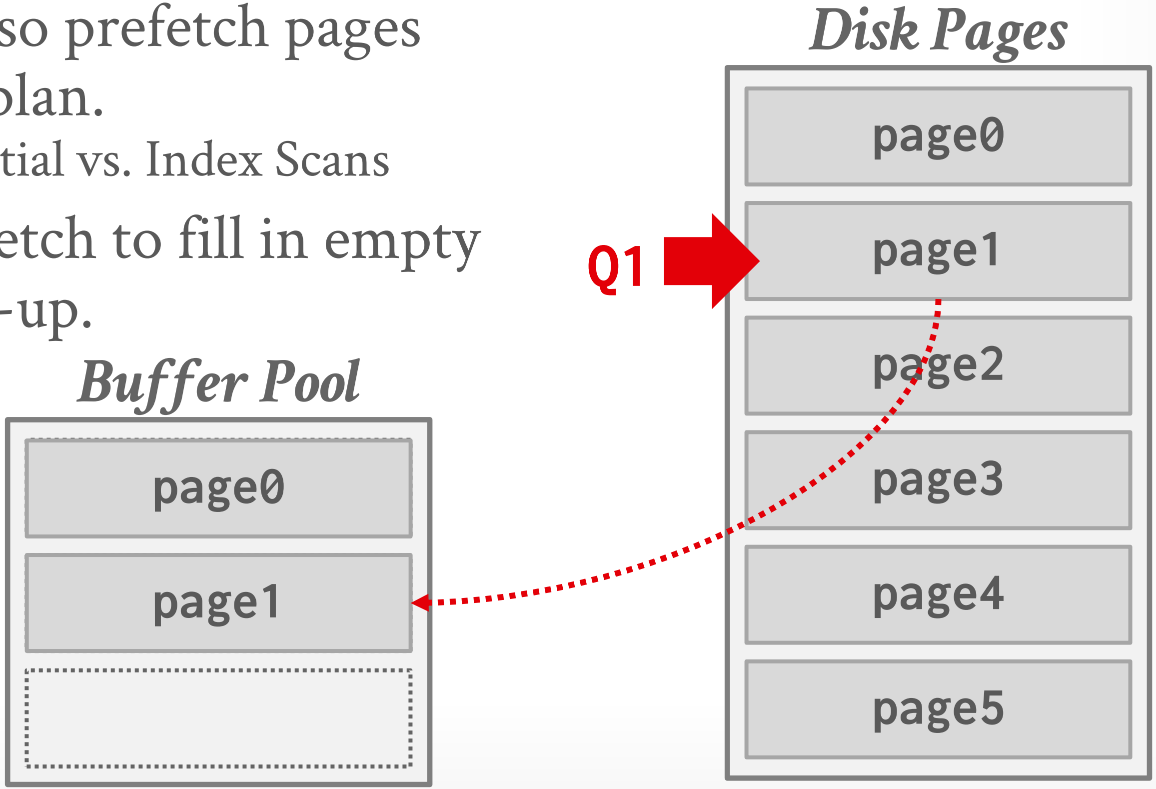
Some DBMS prefetch to fill in empty frames upon start-up.



PRE-FETCHING

The DBMS can also prefetch pages based on a query plan.
→ Examples: Sequential vs. Index Scans

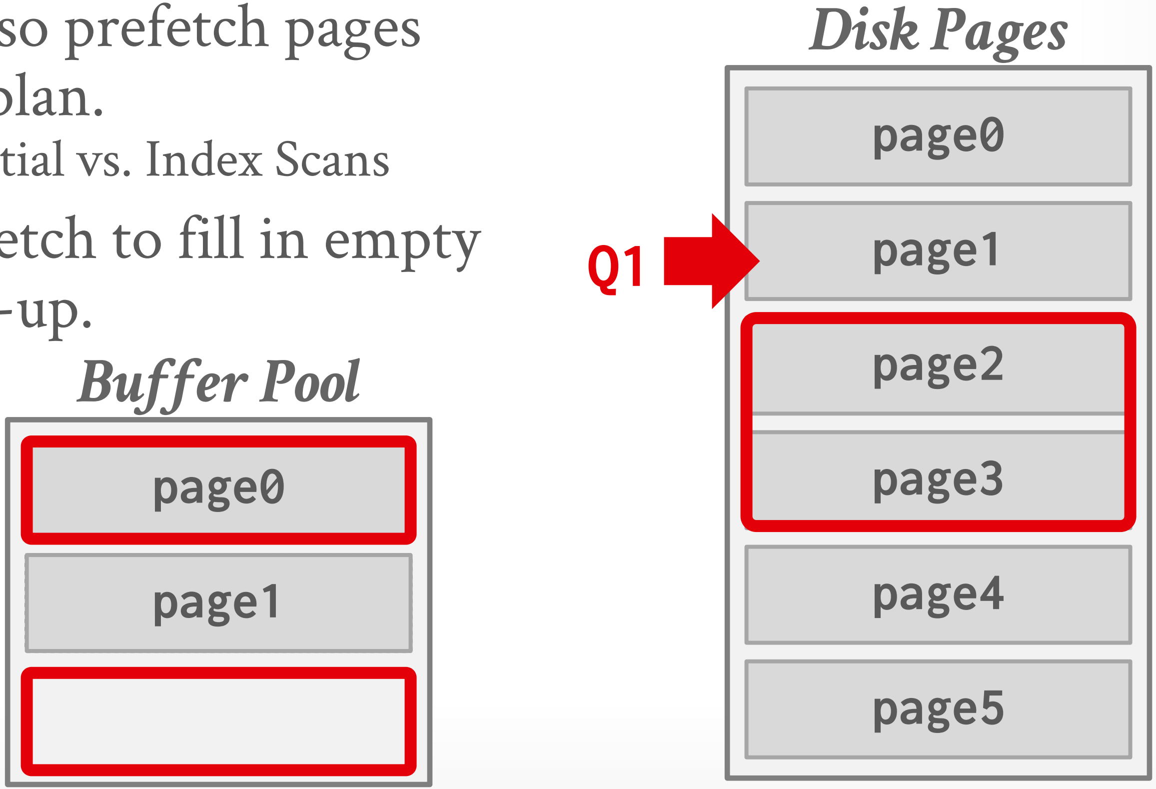
Some DBMS prefetch to fill in empty frames upon start-up.



PRE-FETCHING

The DBMS can also prefetch pages based on a query plan.
→ Examples: Sequential vs. Index Scans

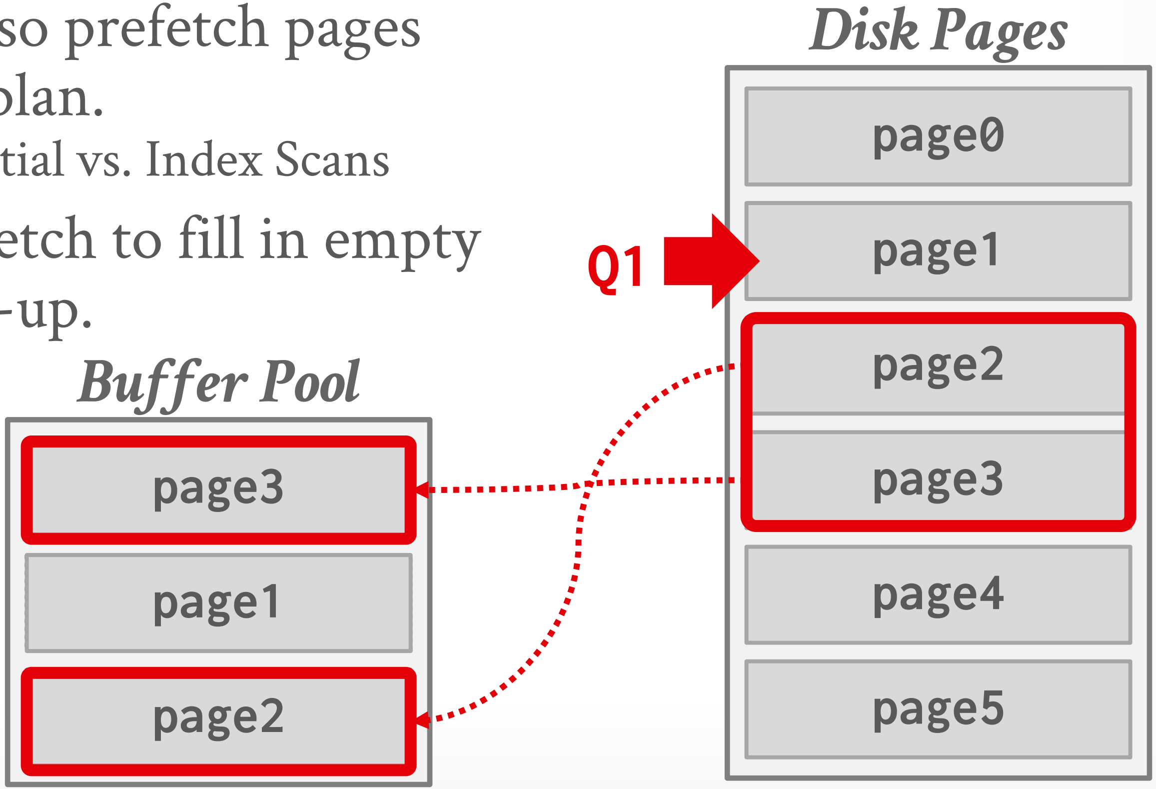
Some DBMS prefetch to fill in empty frames upon start-up.



PRE-FETCHING

The DBMS can also prefetch pages based on a query plan.
→ Examples: Sequential vs. Index Scans

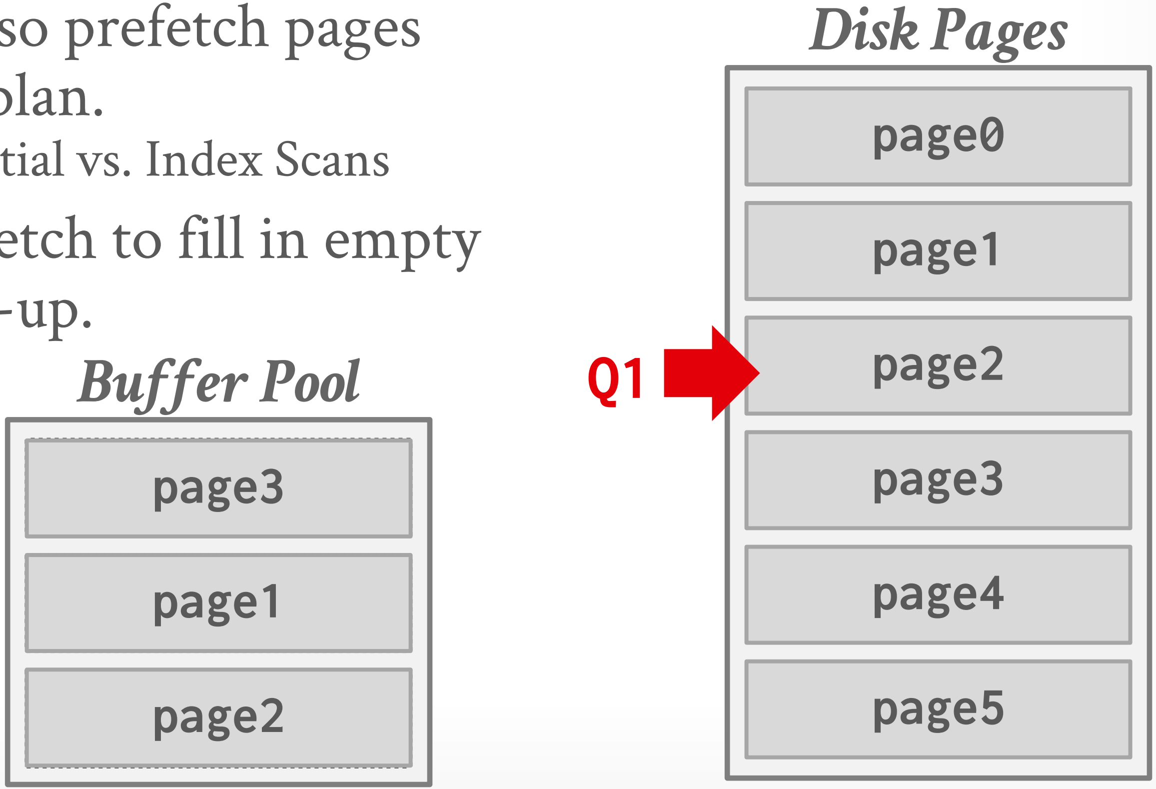
Some DBMS prefetch to fill in empty frames upon start-up.



PRE-FETCHING

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.

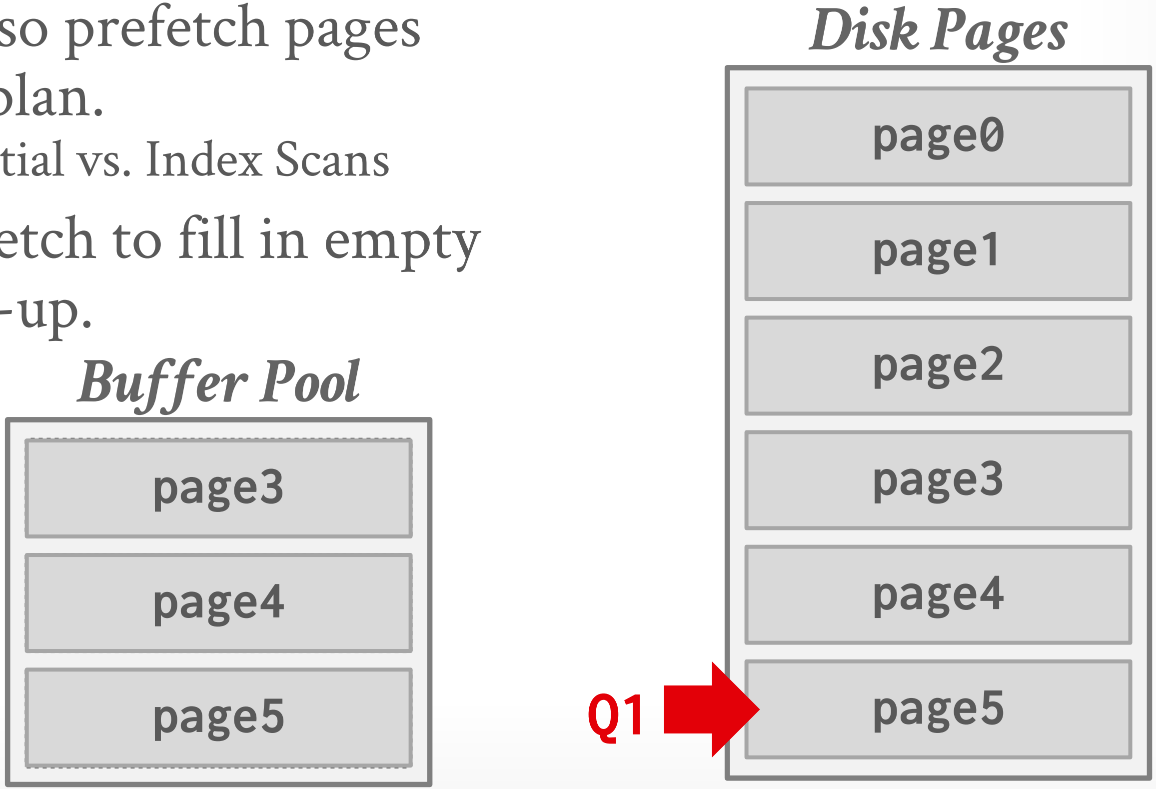


PRE-FETCHING

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

Q1
```
SELECT * FROM A
 WHERE val BETWEEN 100 AND 250
```

Buffer Pool

Disk Pages

index-page0

index-page1

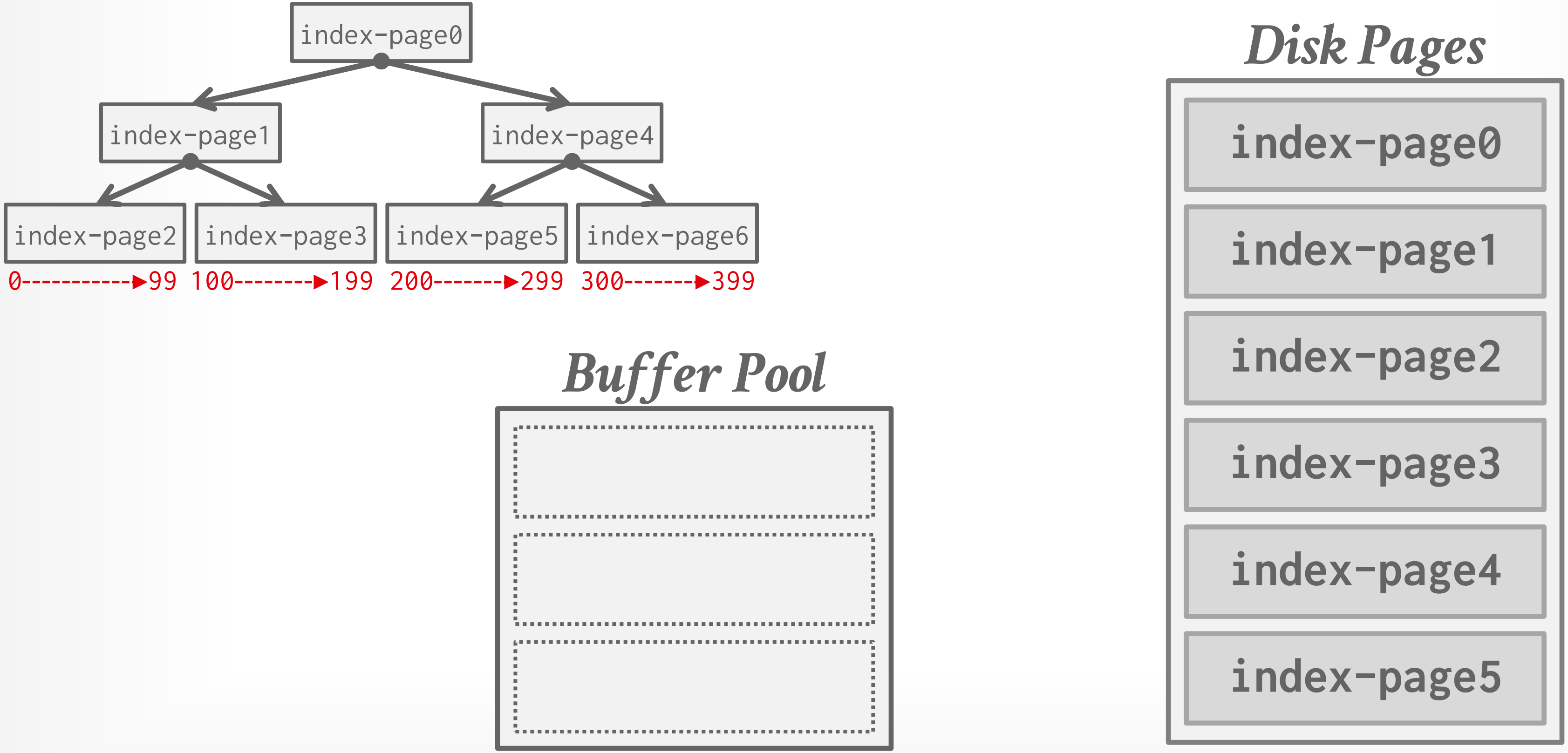
index-page2

index-page3

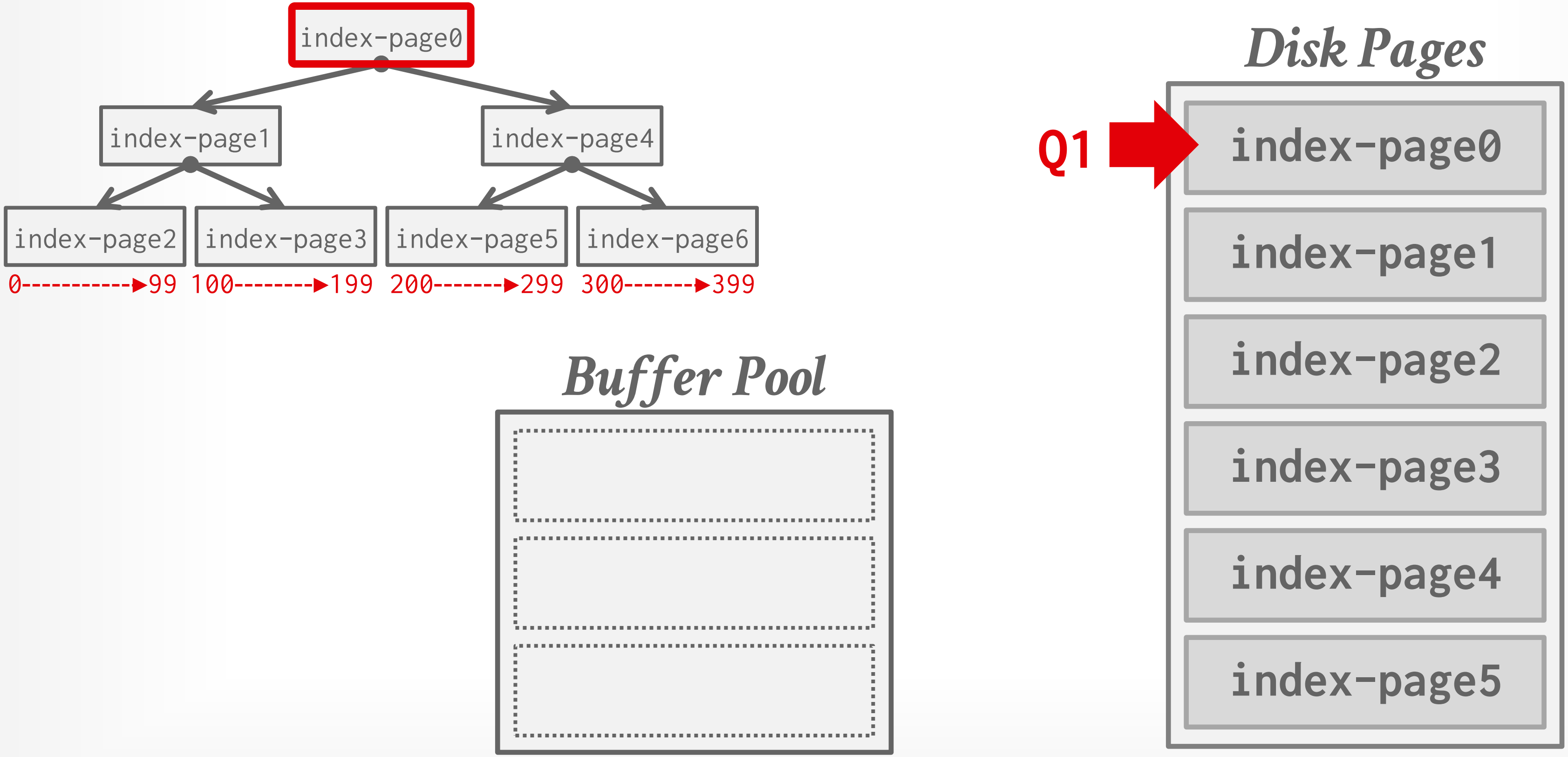
index-page4

index-page5

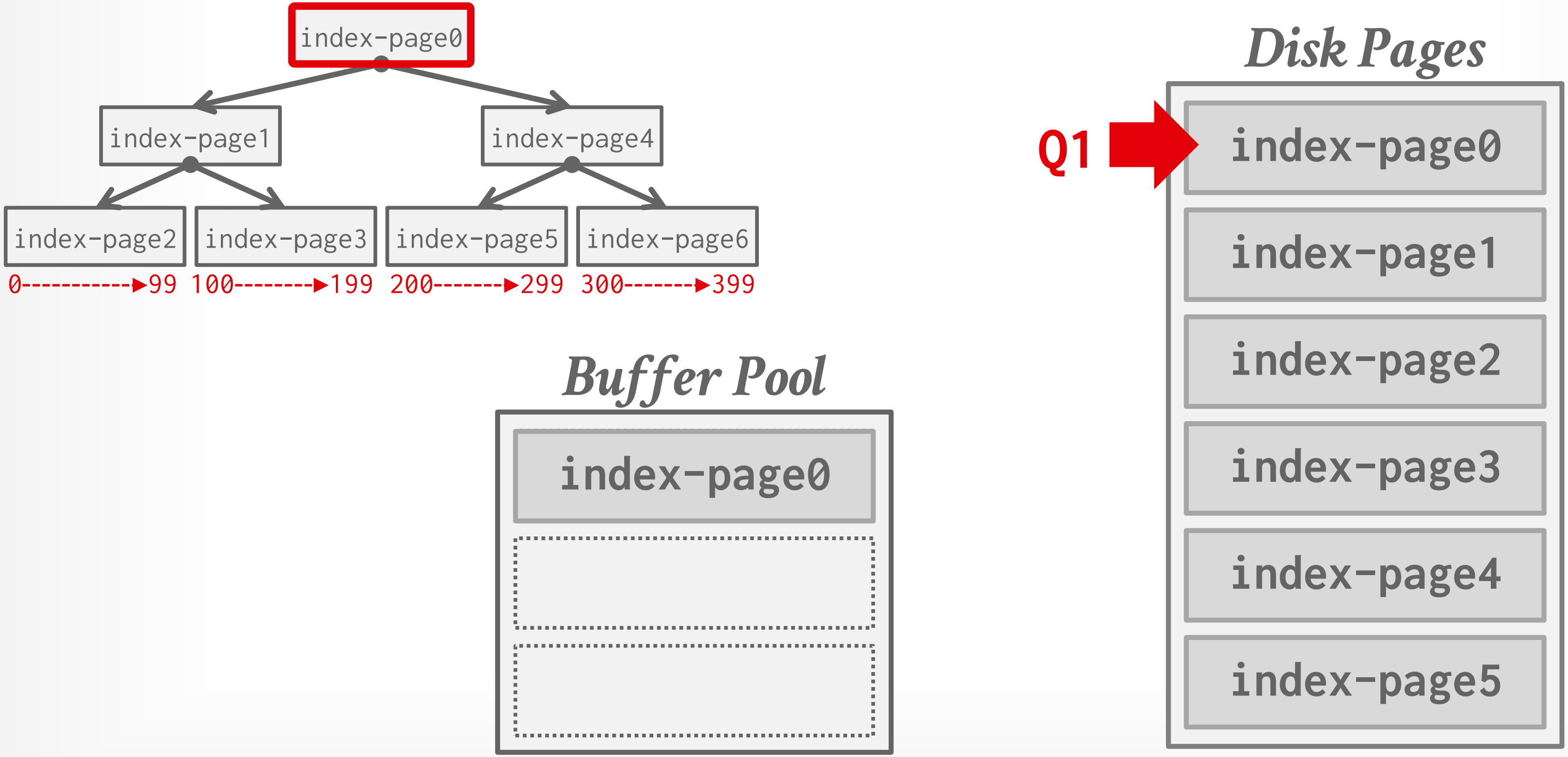
PRE-FETCHING



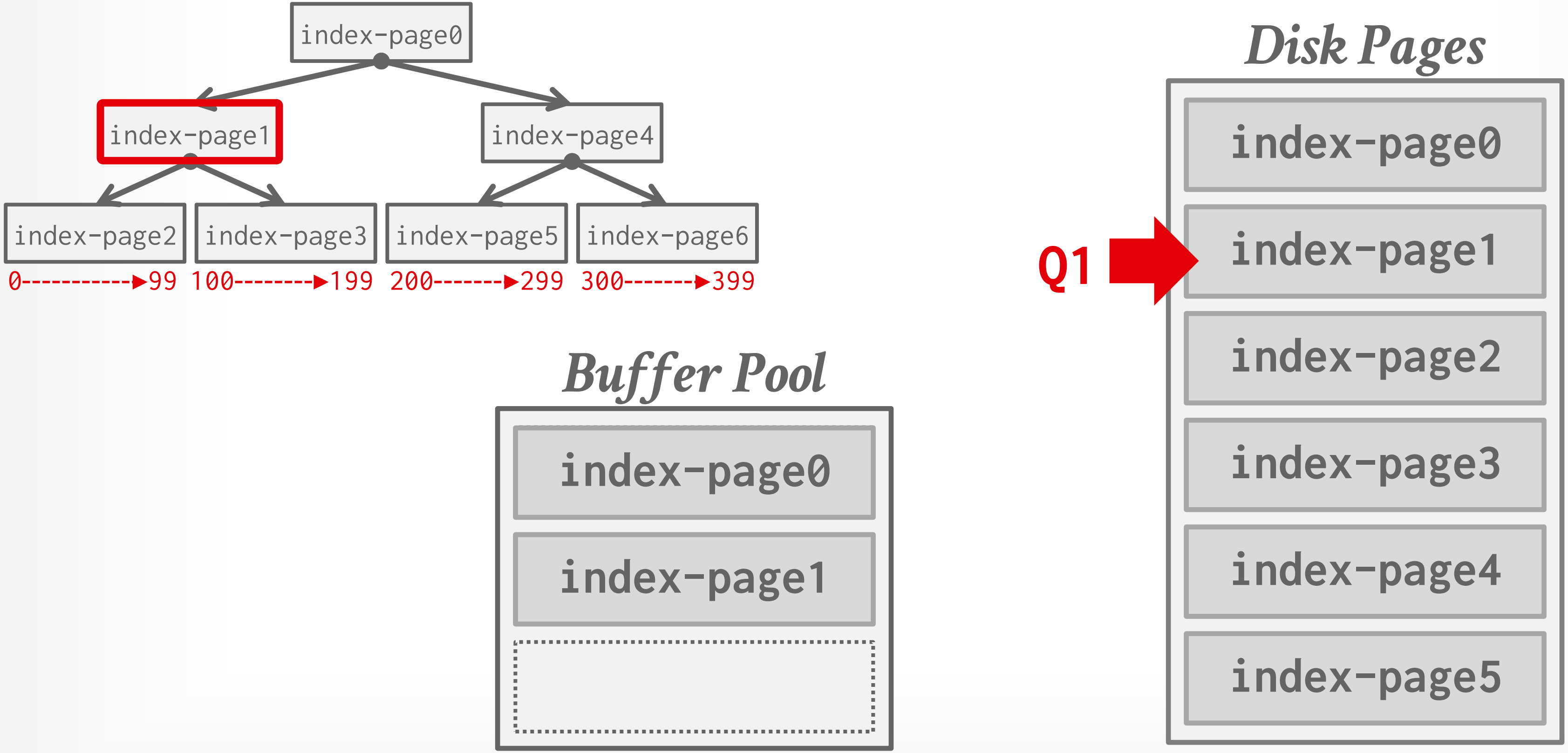
PRE-FETCHING



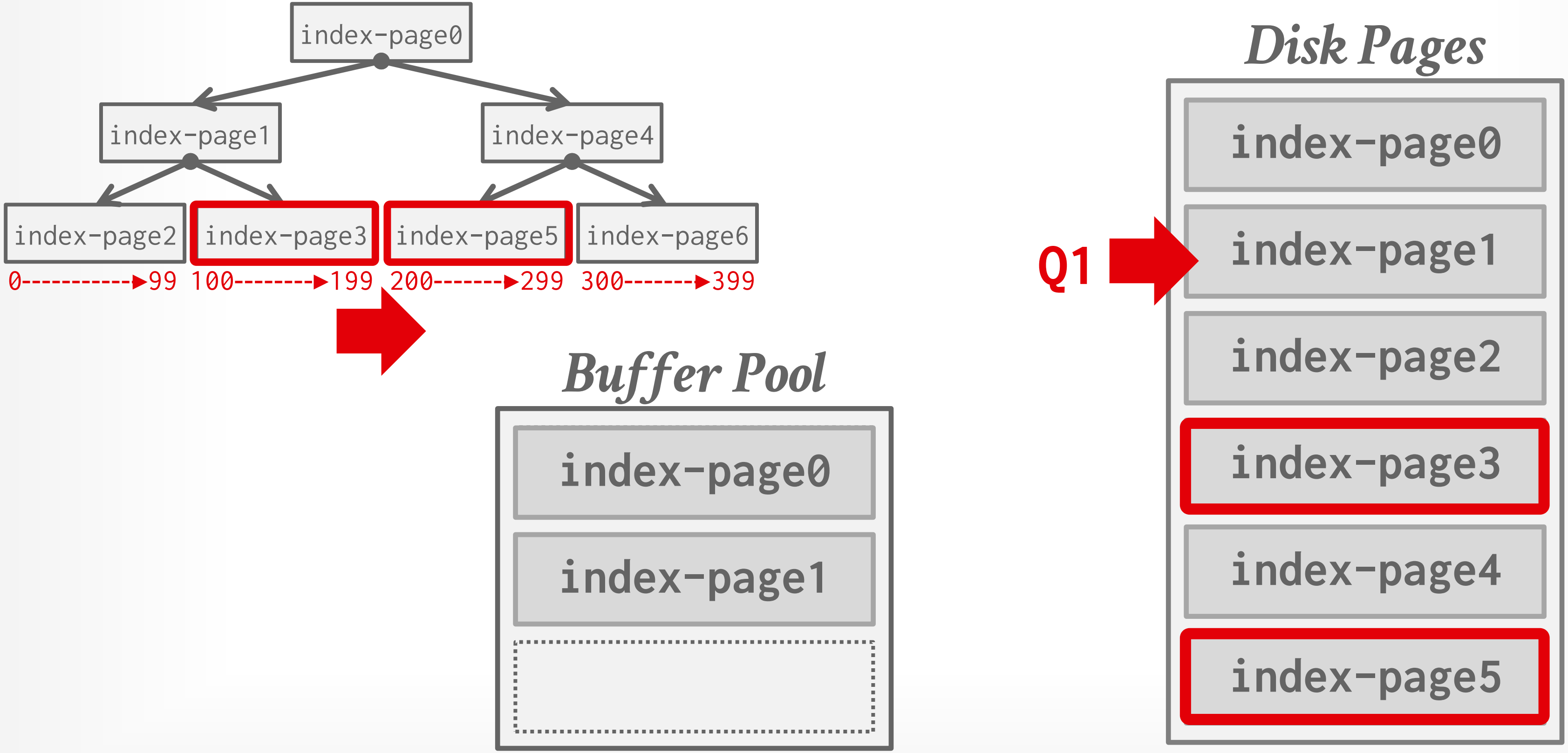
PRE-FETCHING



PRE-FETCHING



PRE-FETCHING



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.

Examples:

→ Fully supported in DB2, MSSQL, Teradata, and Postgres.

→ Oracle only supports cursor sharing for identical queries.

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.

Ex

→

→

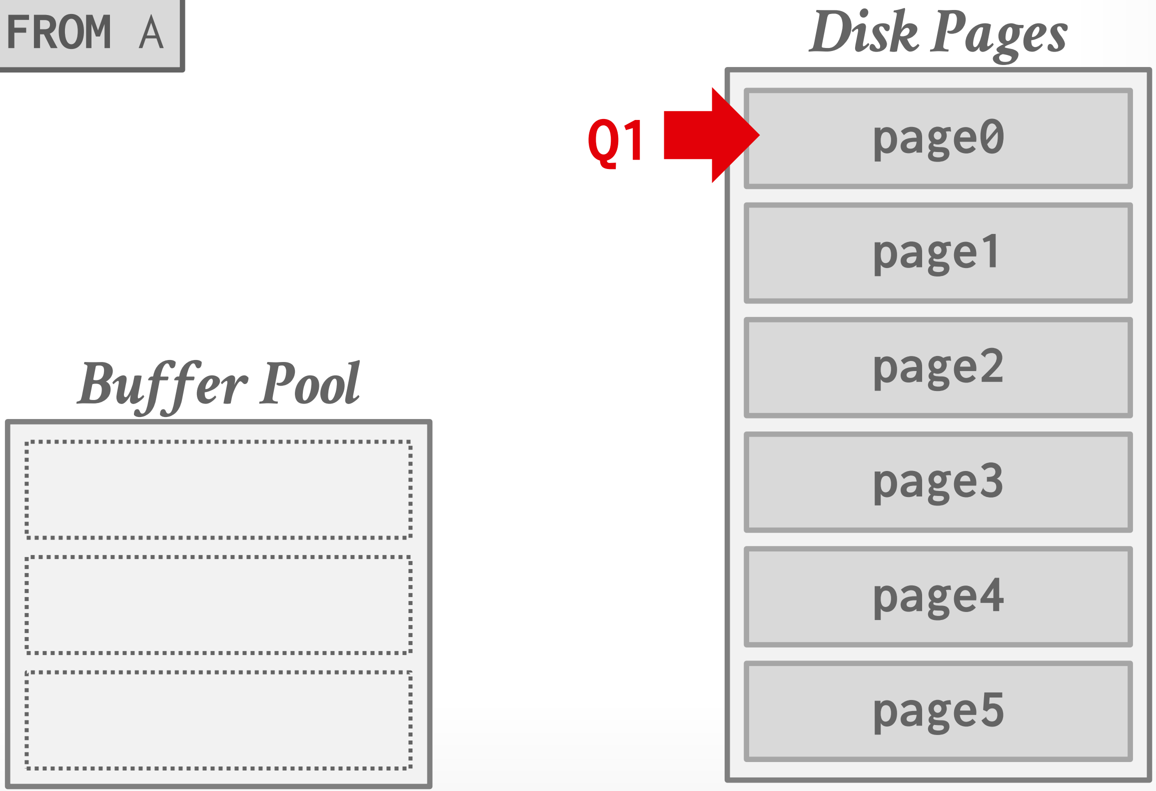
For a textual match to occur, the text of the SQL statements or PL/SQL blocks must be character-for-character identical, including spaces, case, and comments. For example, the following statements cannot use the same shared SQL area:

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

Copy

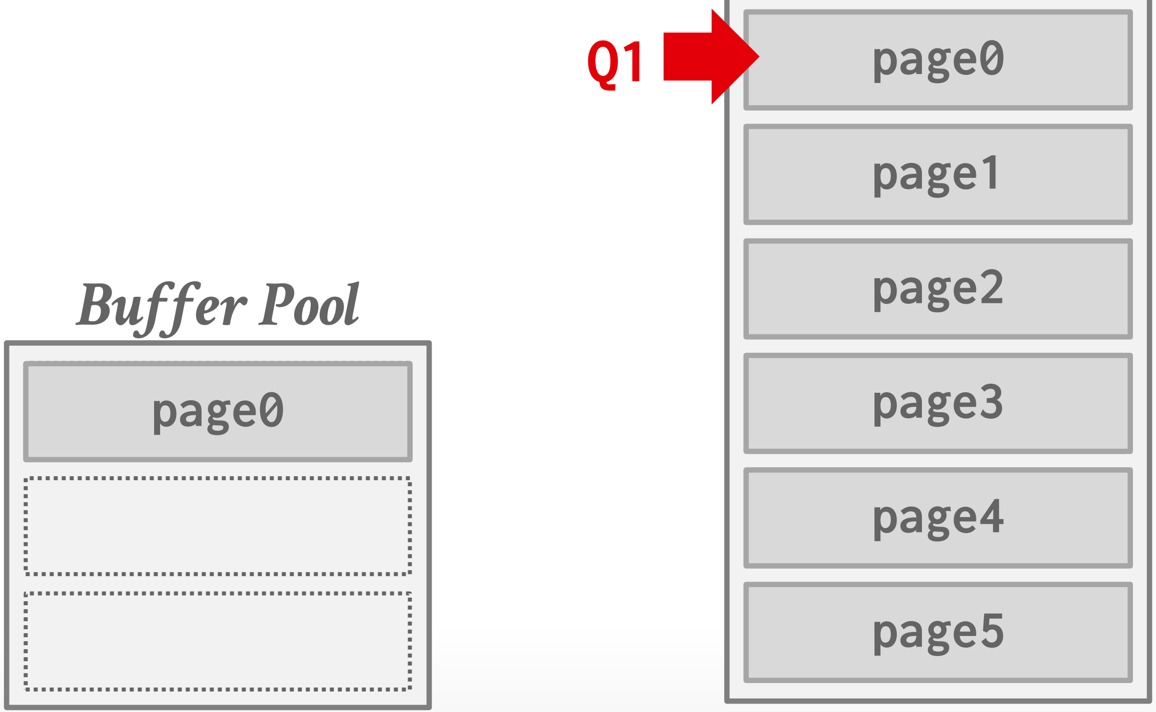
SCAN SHARING

Q1 `SELECT SUM(val) FROM A`



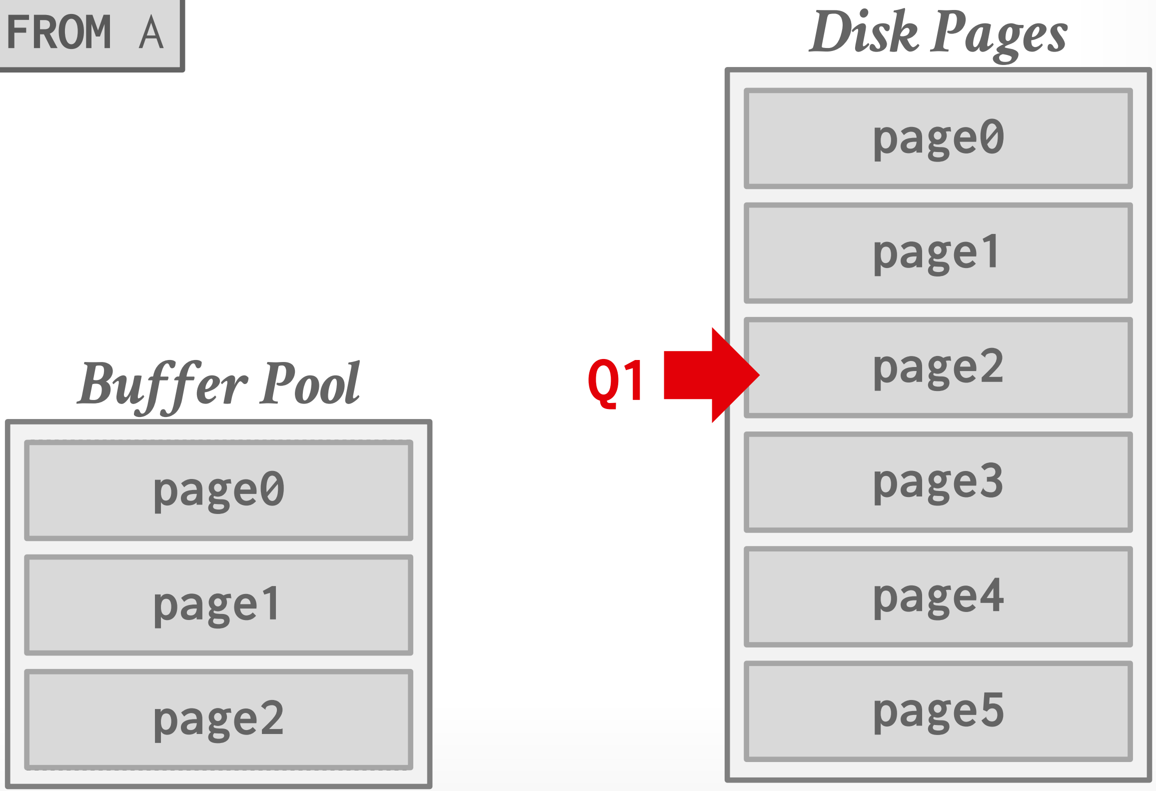
SCAN SHARING

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



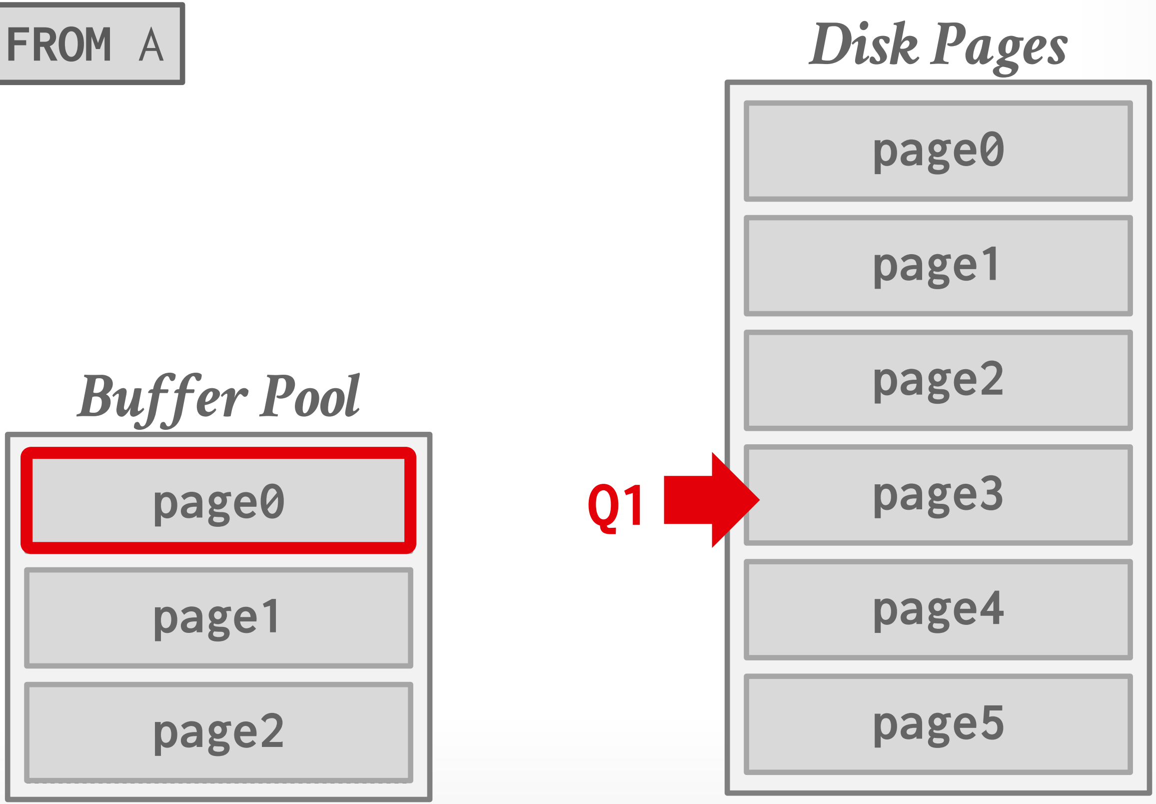
SCAN SHARING

Q1 `SELECT SUM(val) FROM A`



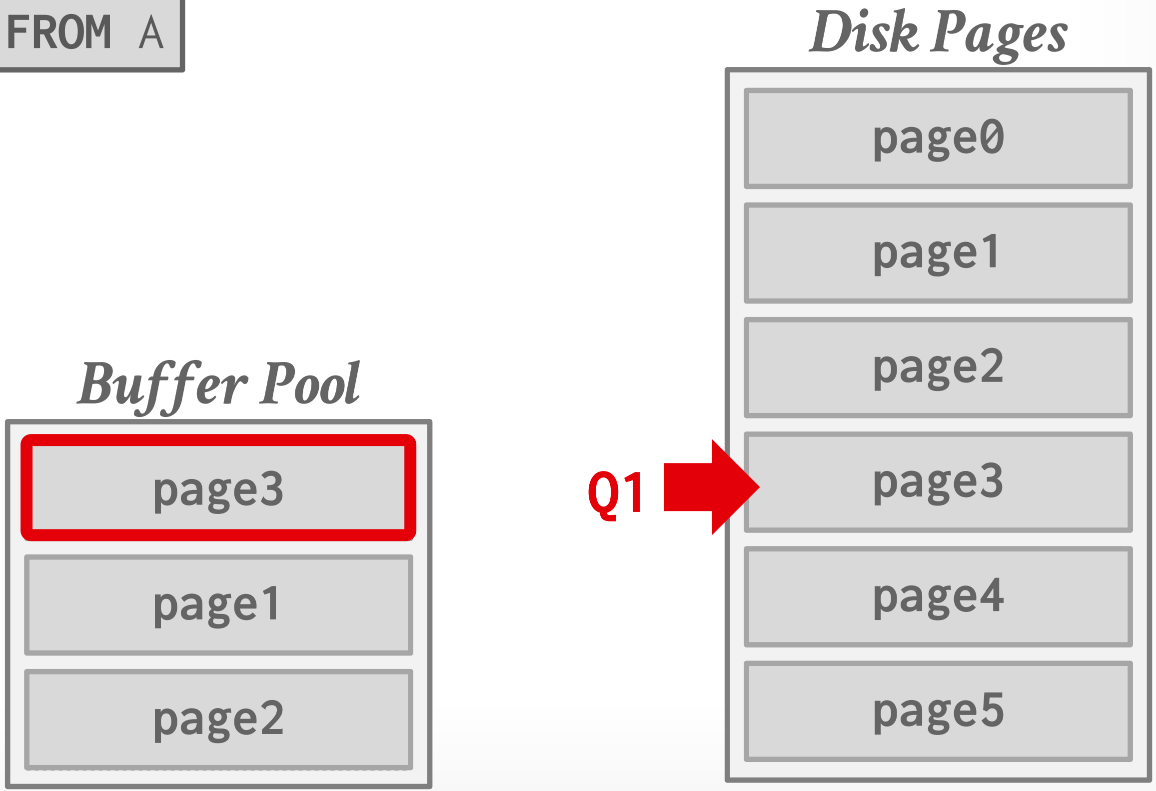
SCAN SHARING

Q1 `SELECT SUM(val) FROM A`



SCAN SHARING

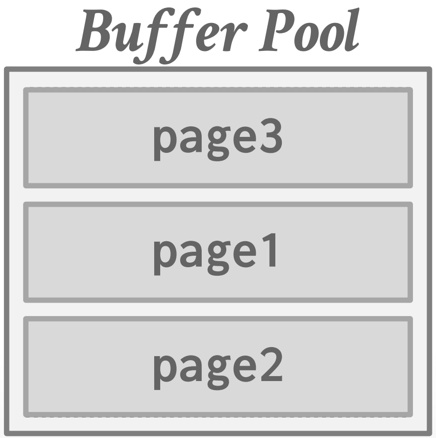
Q1 `SELECT SUM(val) FROM A`

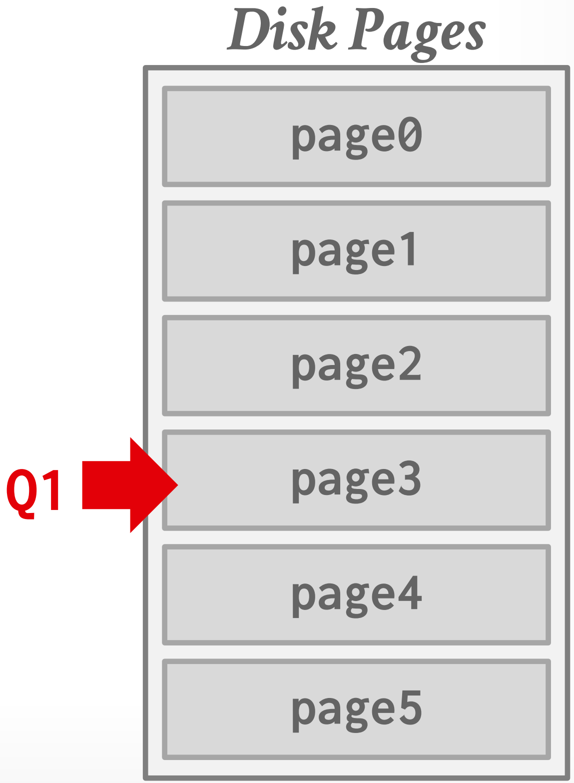


SCAN SHARING

Q1 `SELECT SUM(val) FROM A`

Q2 `SELECT AVG(val) FROM A`

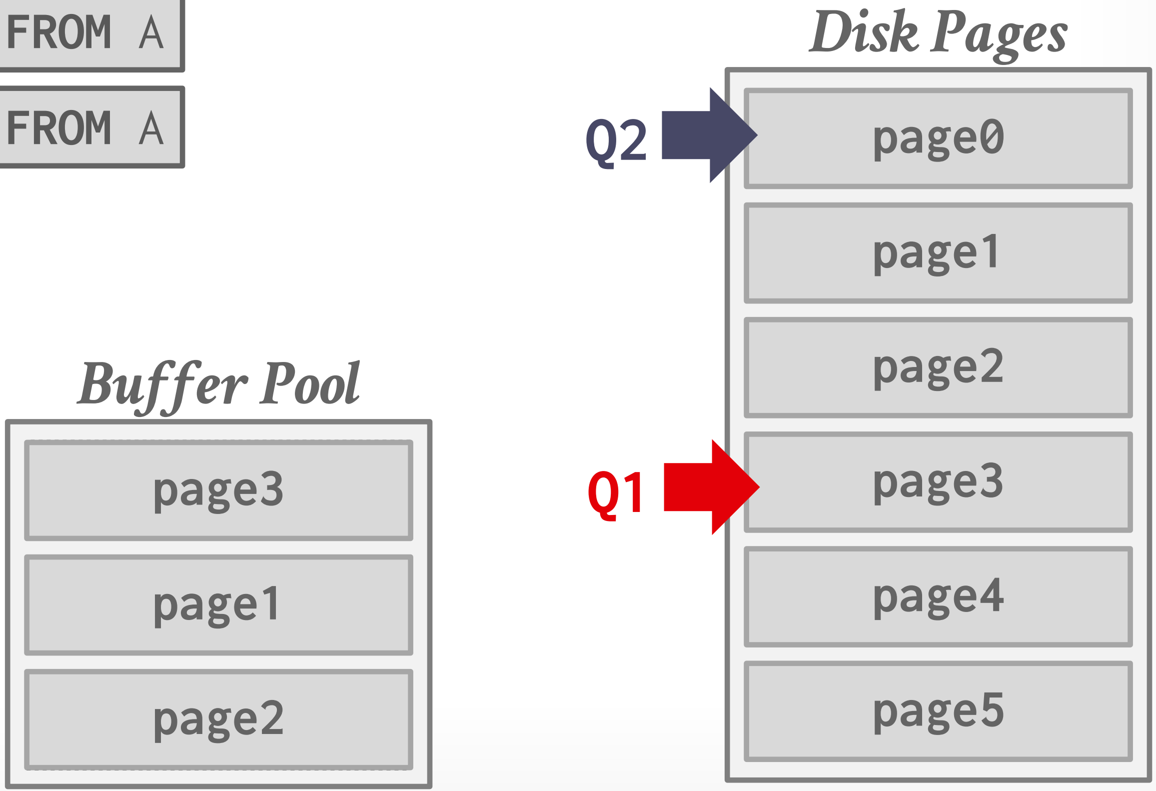




SCAN SHARING

Q1 `SELECT SUM(val) FROM A`

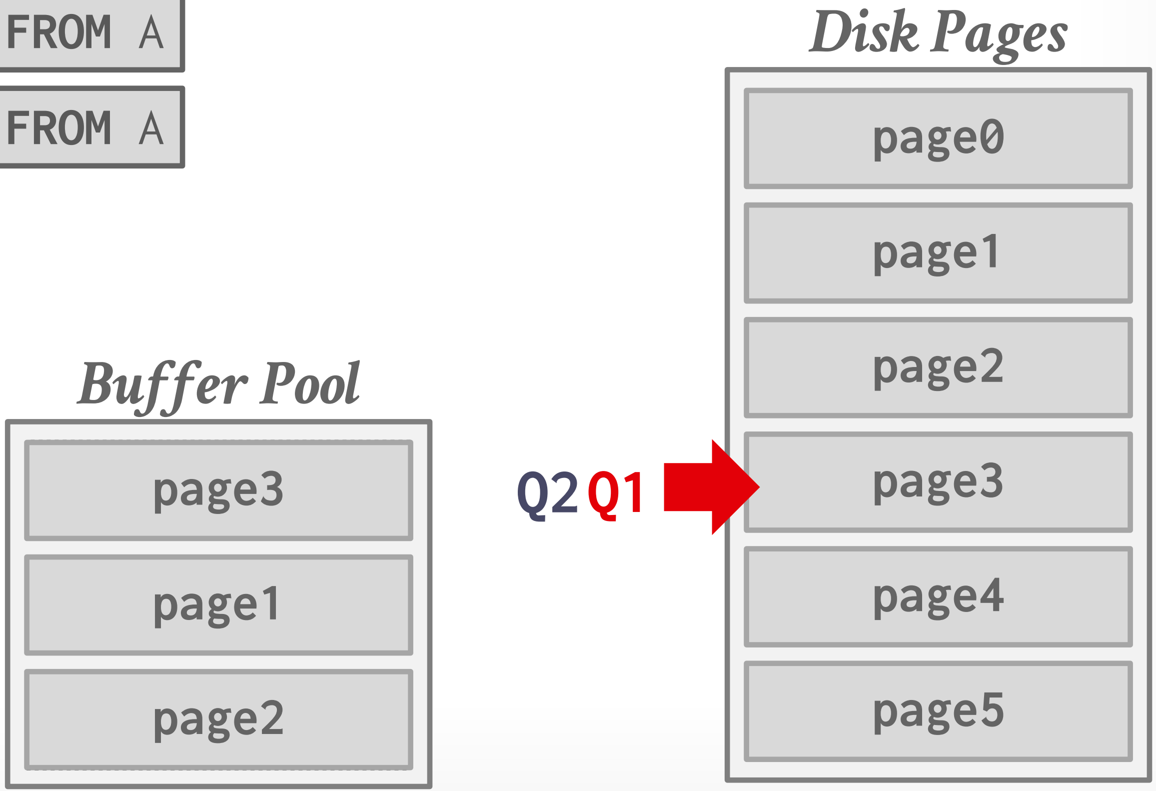
Q2 `SELECT AVG(val) FROM A`



SCAN SHARING

Q1 `SELECT SUM(val) FROM A`

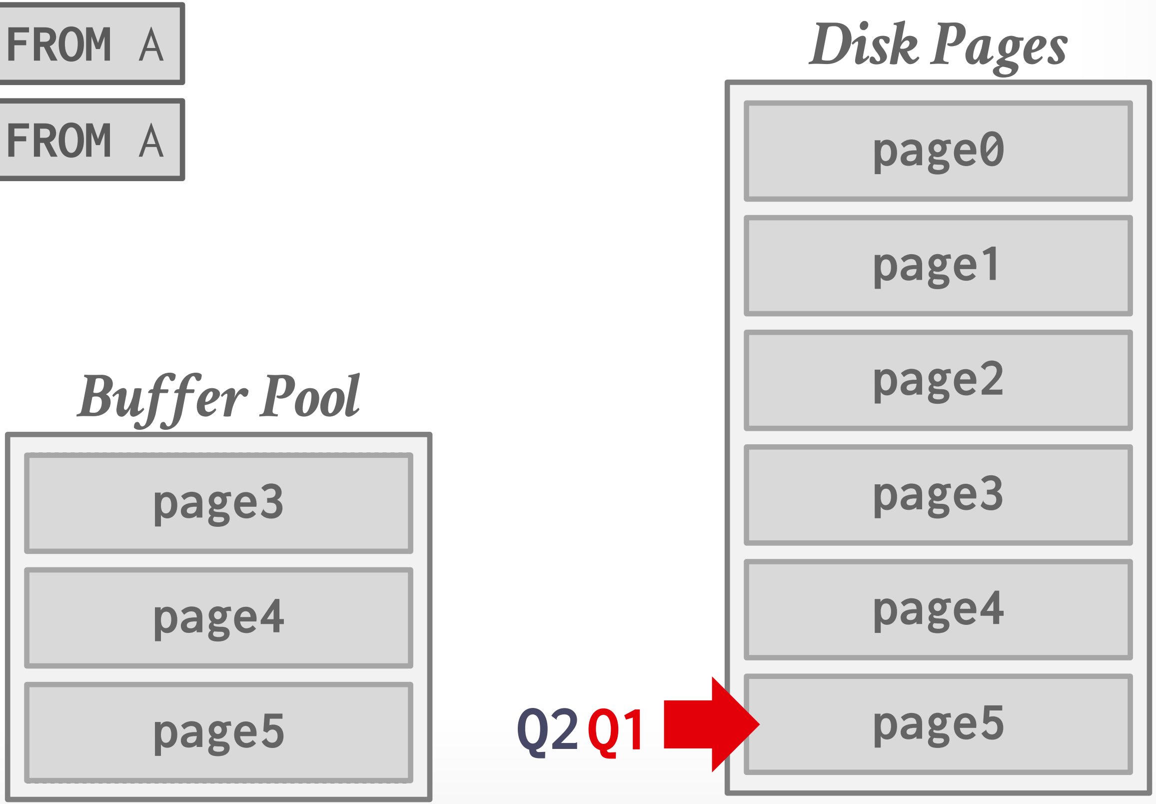
Q2 `SELECT AVG(val) FROM A`



SCAN SHARING

Q1 `SELECT SUM(val) FROM A`

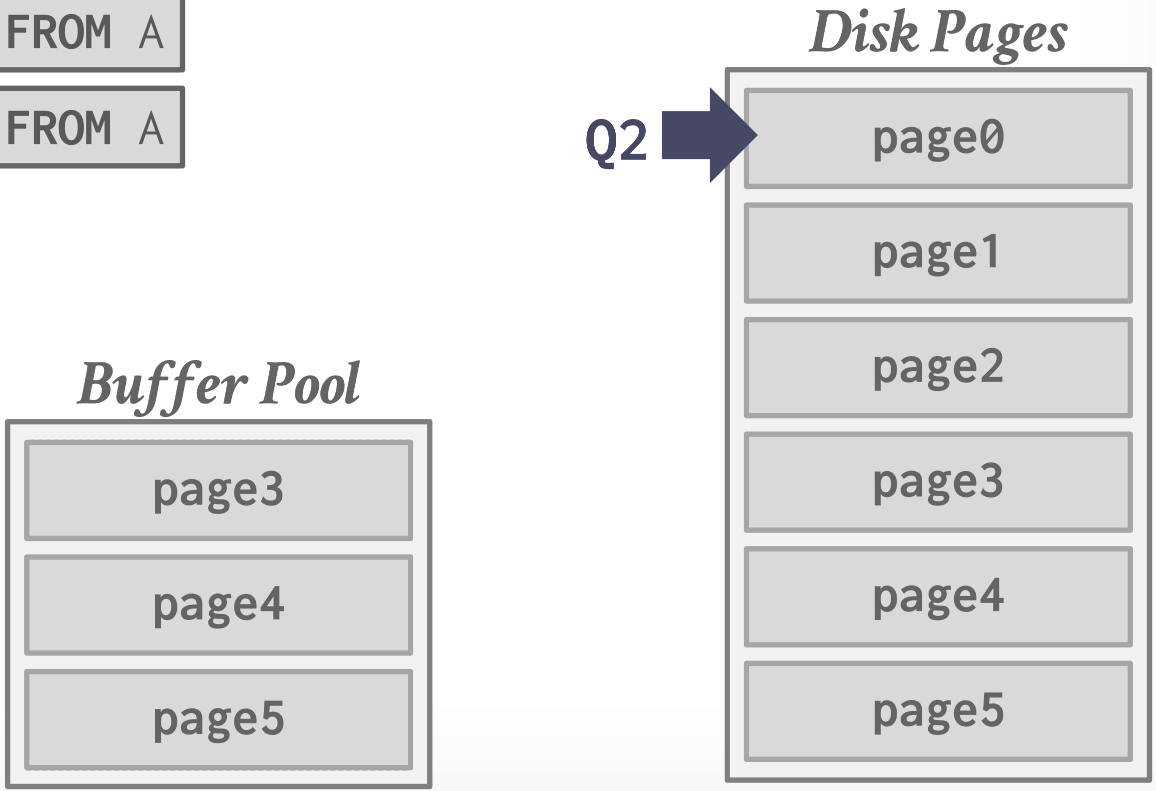
Q2 `SELECT AVG(val) FROM A`



SCAN SHARING

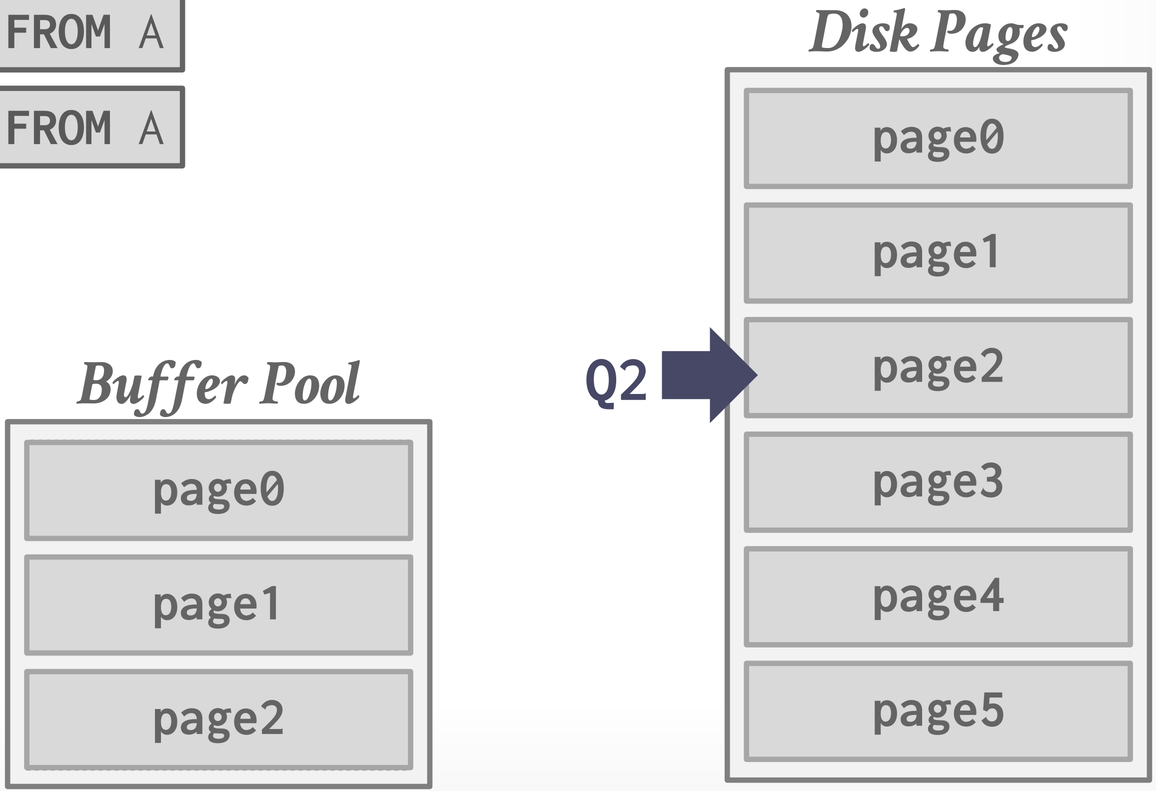
Q1 `SELECT SUM(val) FROM A`

Q2 `SELECT AVG(val) FROM A`



SCAN SHARING

```
Q1  SELECT SUM(val) FROM A
Q2  SELECT AVG(val) FROM A
```



SCAN SHARING

Q1 `SELECT SUM(val) FROM A`

Q2’ `SELECT * FROM A LIMIT 100`

Buffer Pool

- page0
- page1
- page2

Disk Pages

- page0
- page1
- page2
- page3
- page4
- page5

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.

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

Back to Storage Structures!

Log-Structured Storage

Index-Organized Storage

Catalogs