Carnegie Mellon University

Intro to Database Systems (15-445/645)

Lecture #03

Database Storage *Part 1*

FALL 2023 » Prof. Andy Pavlo • Prof. Jignesh Patel

ADMINISTRIVIA

Homework #1 is due September 10th @ 11:59pm

Project #0 is due September 10th @ 11:59pm

Project #1 will be released on September 8th

LAST CLASS

We now understand what a database looks like at a logical level and how to write queries to read/write data (e.g., using SQL).

We will next learn how to build software that manages a database (i.e., a DBMS).

COURSE OUTLINE

Relational Databases
Storage
Execution
Concurrency Control
Recovery
Distributed Databases
Potpourri

Query Planning
Operator Execution
Access Methods
Buffer Pool Manager
Disk Manager

DISK-BASED ARCHITECTURE

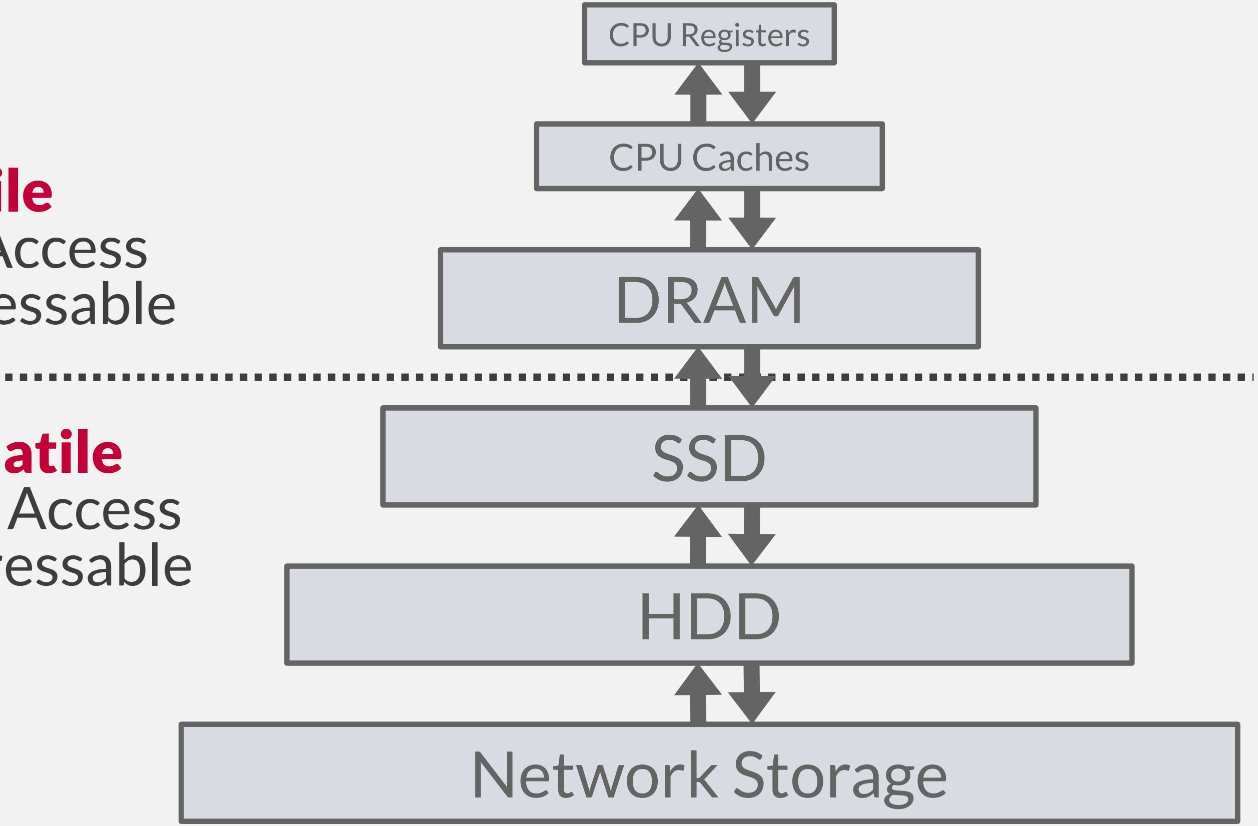
The DBMS assumes that the primary storage location of the database is on non-volatile disk.

The DBMS's components manage the movement of data between non-volatile and volatile storage.

STORAGE HIERARCHY

Faster
Smaller
Expensive

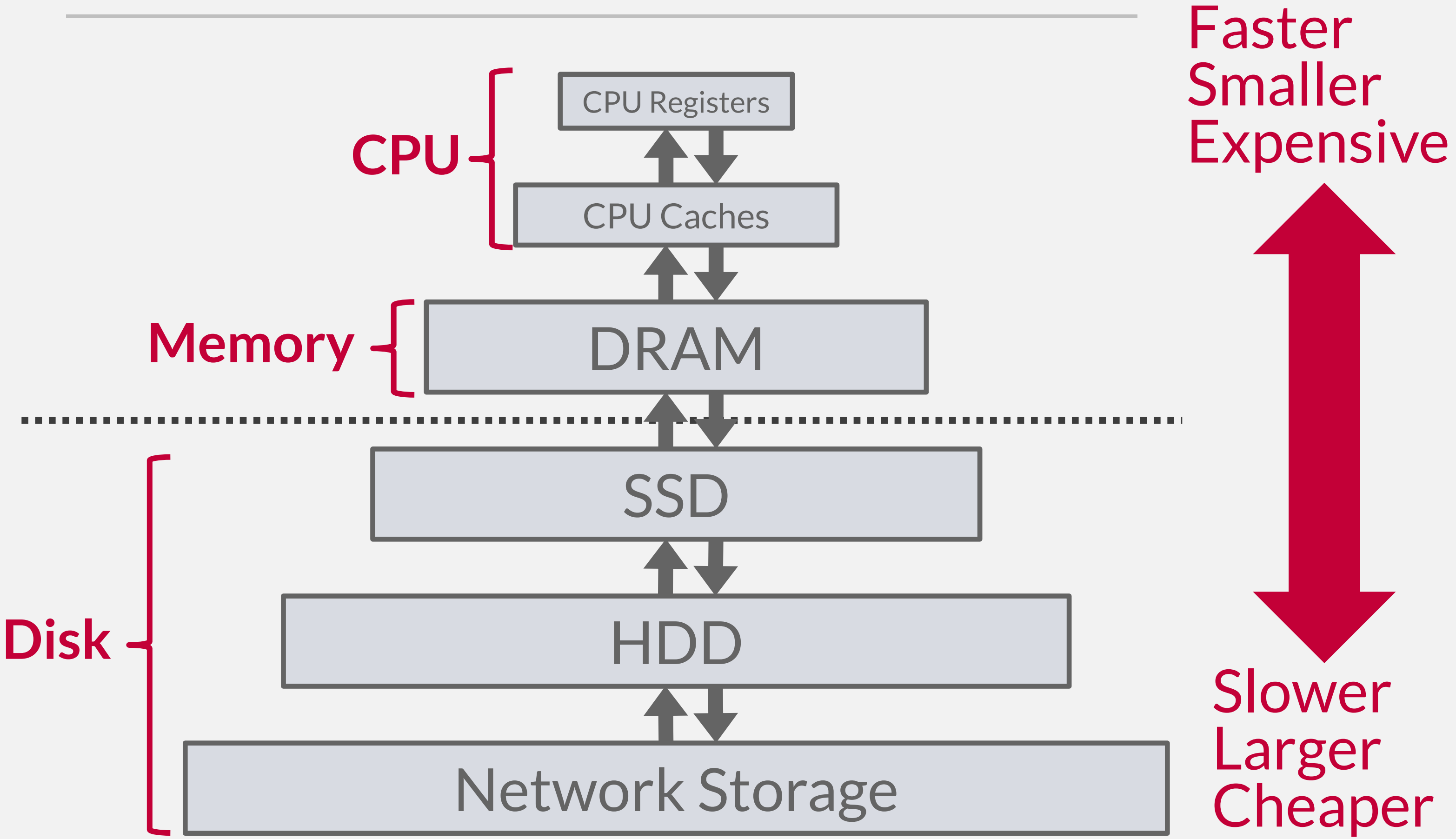
Volatile
Random Access
Byte-Addressable



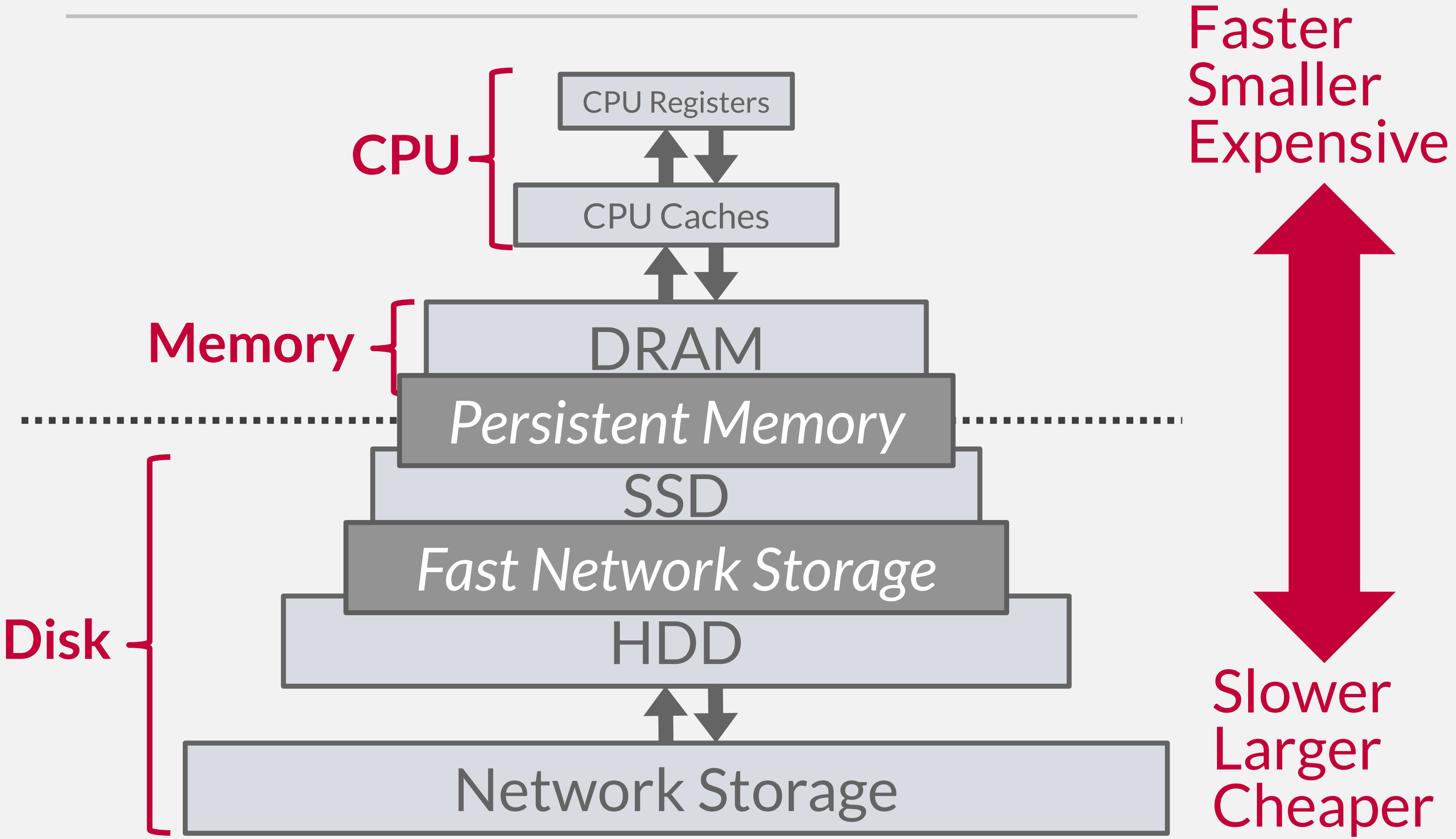
Non-Volatile
Sequential Access
Block-Addressable

Slower
Larger
Cheaper

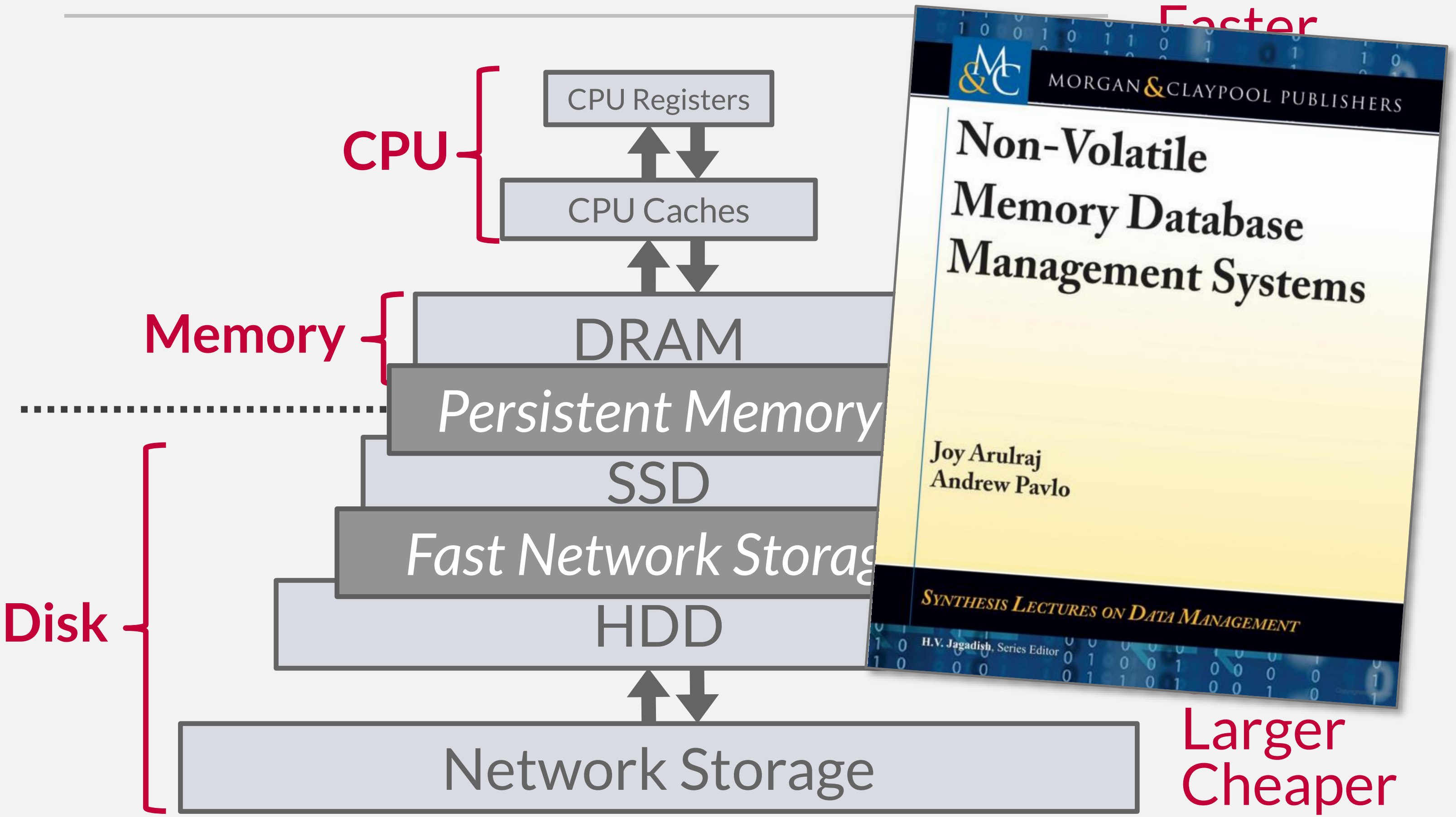
STORAGE HIERARCHY



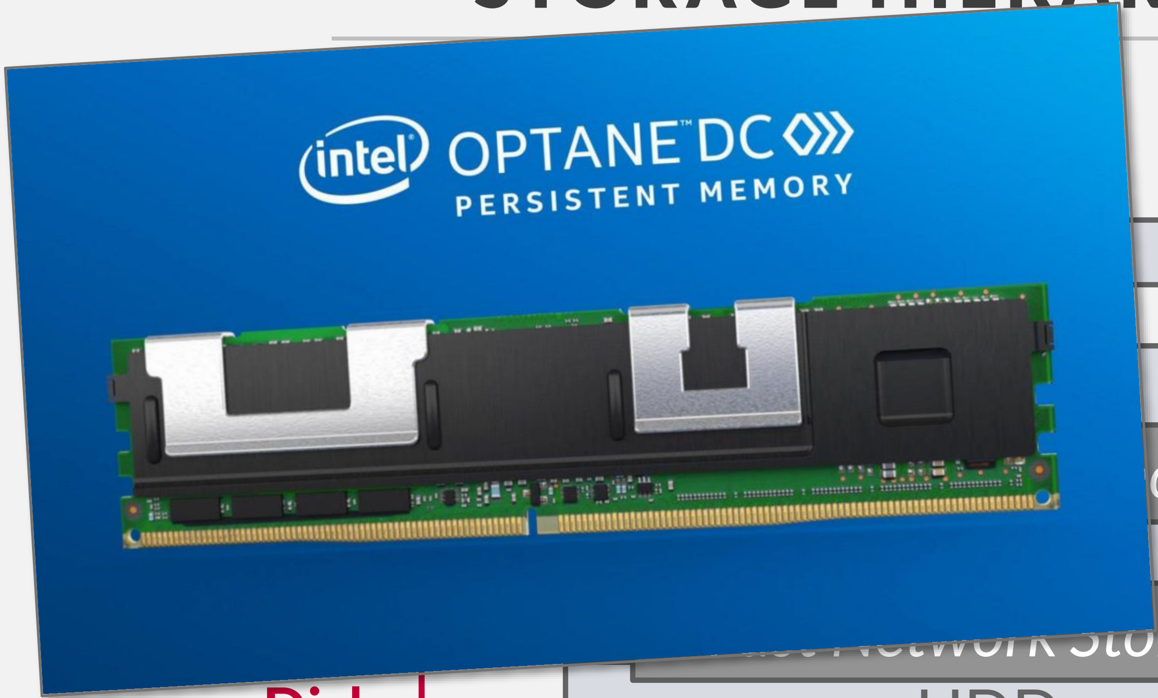
STORAGE HIERARCHY

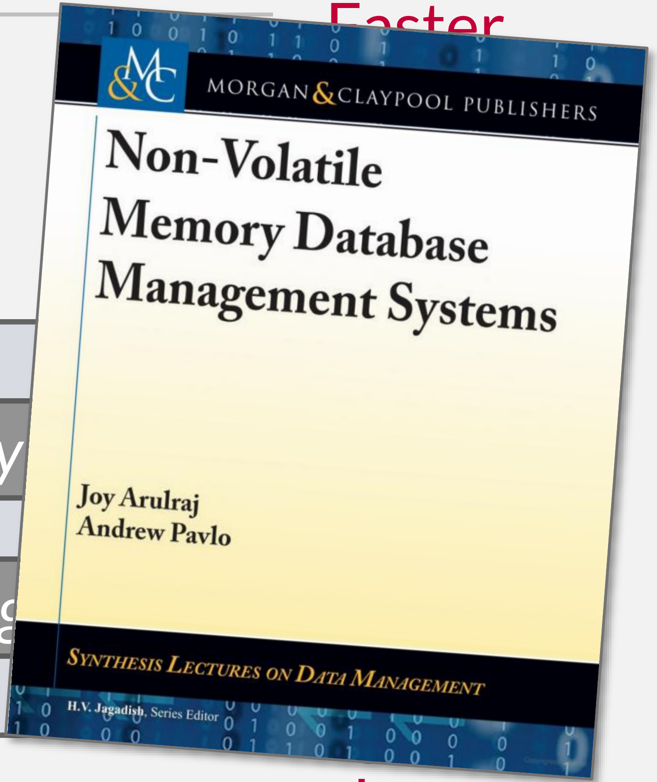


STORAGE HIERARCHY



STORAGE HIERARCHY





Faster

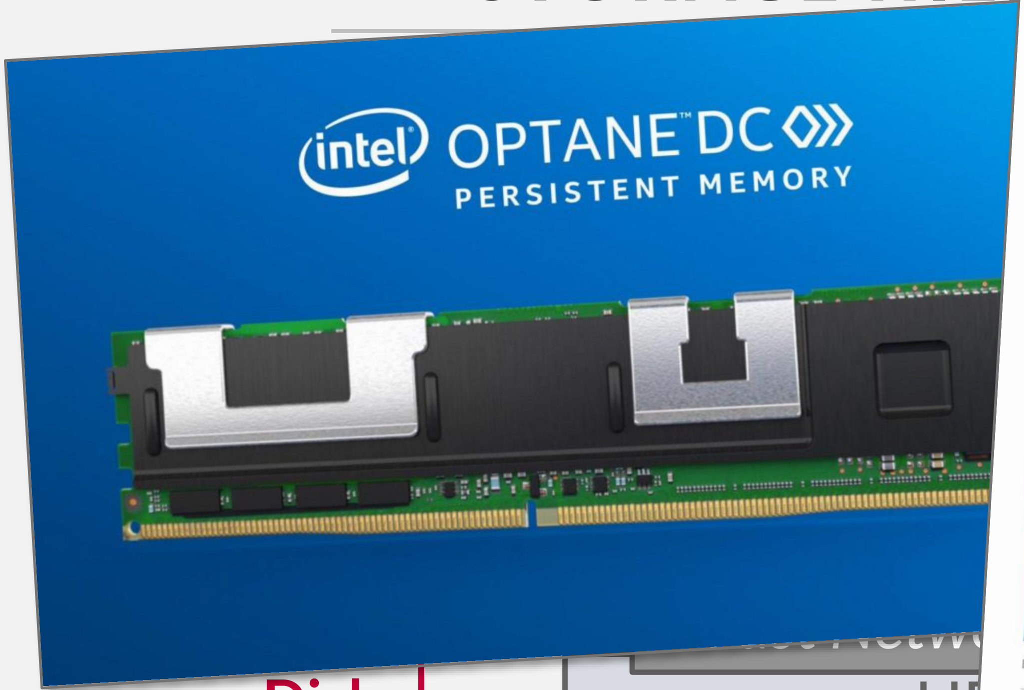
Disk

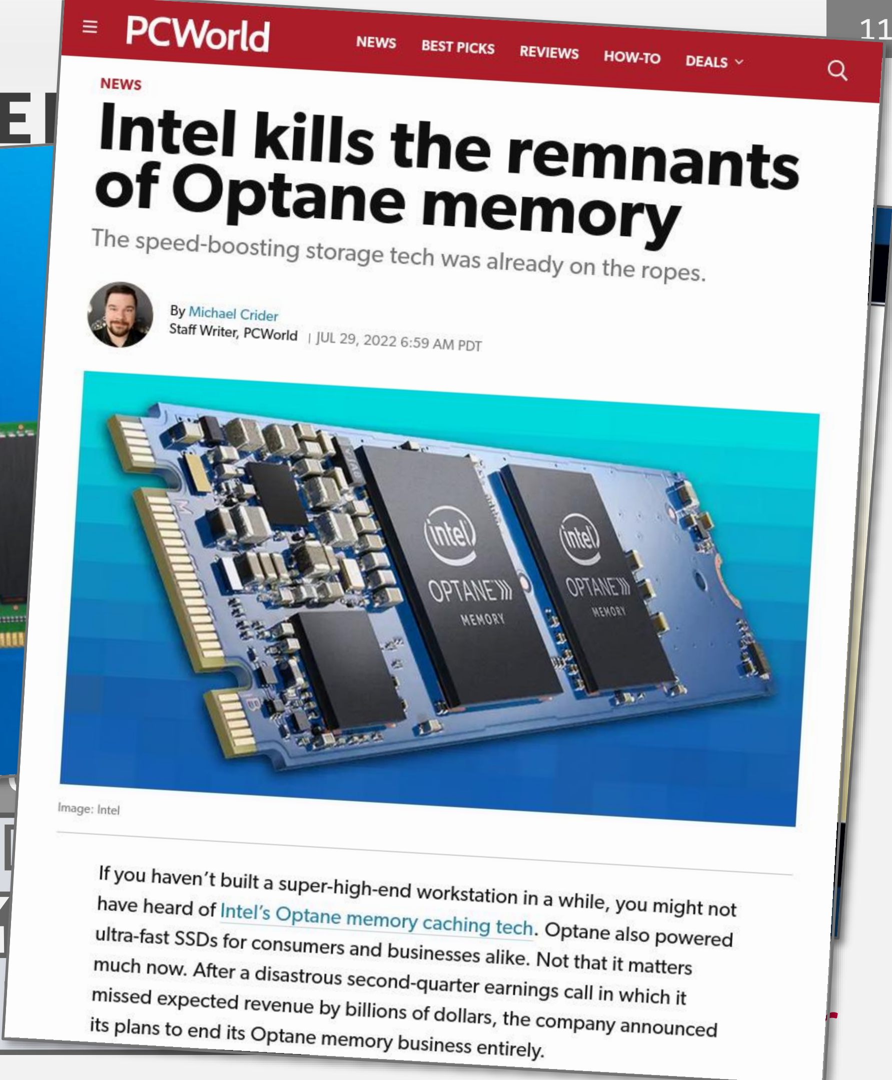
HDD

Network Storage

Larger
Cheaper

STORAGE HIE





NEWS

Intel kills the remnants of Optane memory

The speed-boosting storage tech was already on the ropes.

By Michael Crider
Staff Writer, PCWorld | JUL 29, 2022 6:59 AM PDT

Image: Intel

If you haven't built a super-high-end workstation in a while, you might not have heard of Intel's Optane memory caching tech. Optane also powered ultra-fast SSDs for consumers and businesses alike. Not that it matters much now. After a disastrous second-quarter earnings call in which it missed expected revenue by billions of dollars, the company announced its plans to end its Optane memory business entirely.

Disk

HD

Networ

ACCESS TIMES

Latency Numbers Every Programmer Should Know

1 ns	L1 Cache Ref
4 ns	L2 Cache Ref
100 ns	DRAM
16,000 ns	SSD
2,000,000 ns	HDD
~50,000,000 ns	Network Storage
1,000,000,000 ns	Tape Archives

Source: Colin Scott

ACCESS TIMES

Latency Numbers Every Programmer Should Know

1 ns	L1 Cache Ref	←	**1 sec**
4 ns	L2 Cache Ref	←	**4 sec**
100 ns	DRAM	←	**100 sec**
16,000 ns	SSD	←	**4.4 hours**
2,000,000 ns	HDD	←	**3.3 weeks**
~50,000,000 ns	Network Storage	←	**1.5 years**
1,000,000,000 ns	Tape Archives	←	**31.7 years**

Source: Colin Scott

SEQUENTIAL VS. RANDOM ACCESS

Random access on non-volatile storage is almost always much slower than sequential access.

DBMS will want to maximize sequential access.

→ Algorithms try to reduce number of writes to random pages so that data is stored in contiguous blocks.
→ Allocating multiple pages at the same time is called an extent.

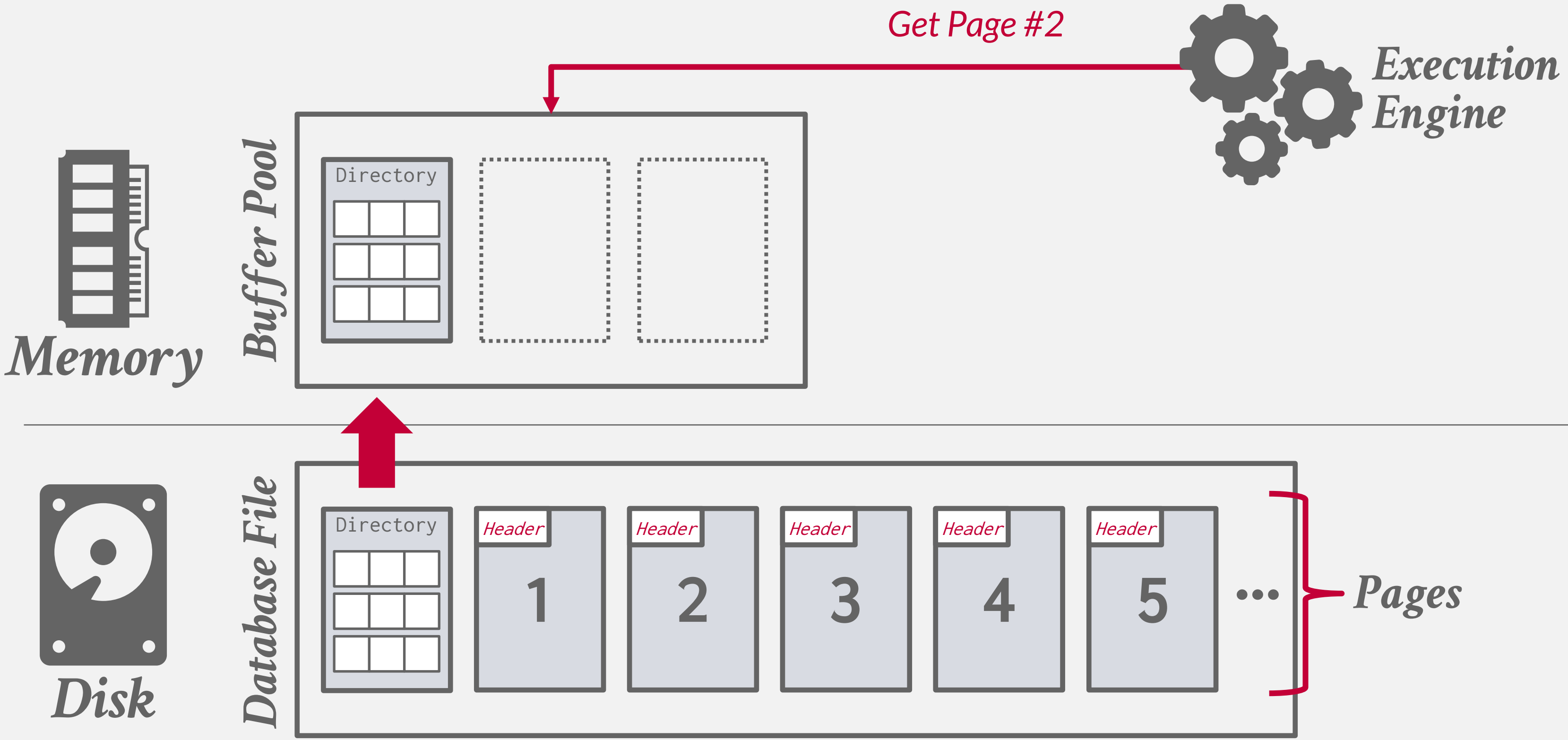
SYSTEM DESIGN GOALS

Allow the DBMS to manage databases that exceed the amount of memory available.

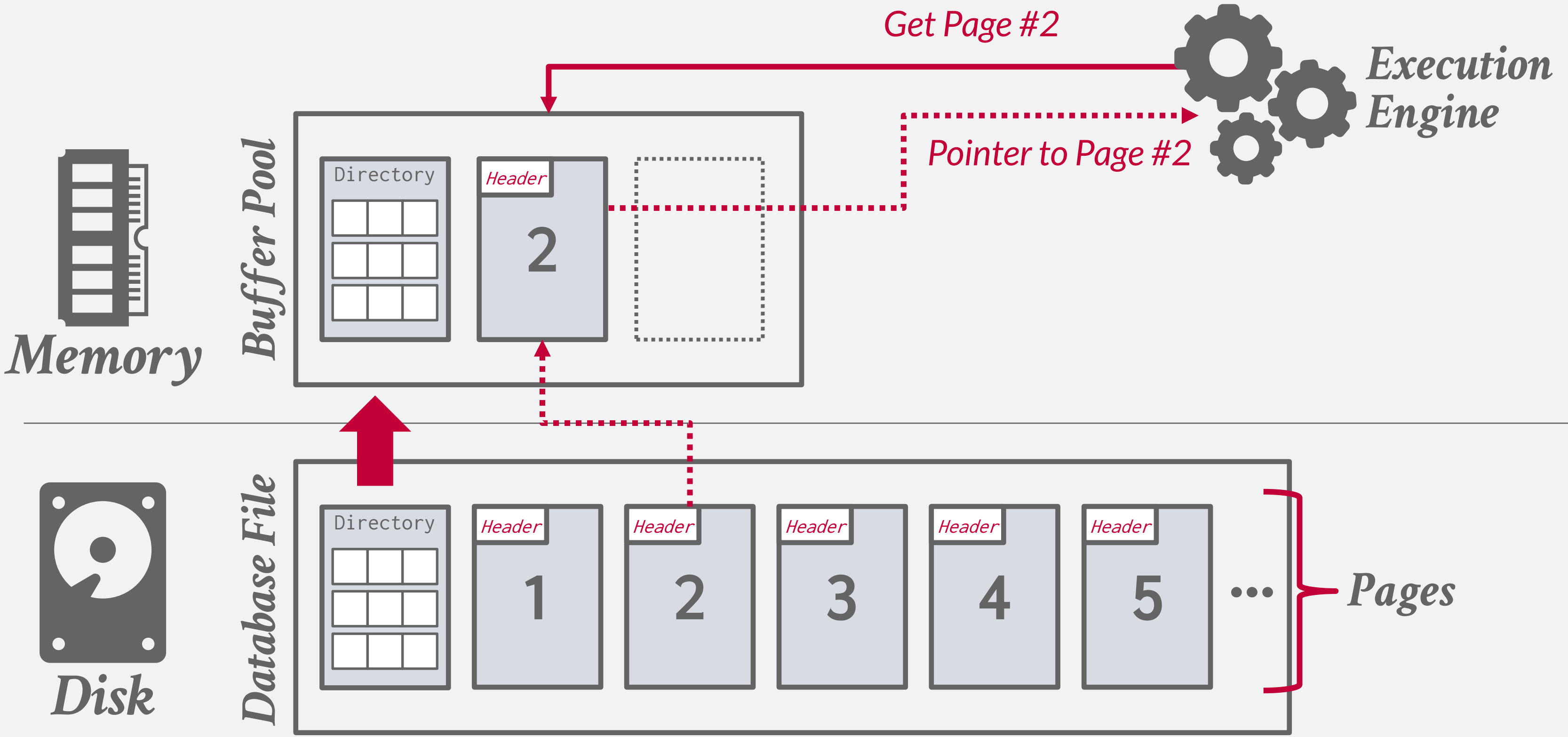
Reading/writing to disk is expensive, so it must be managed carefully to avoid large stalls and performance degradation.

Random access on disk is usually much slower than sequential access, so the DBMS will want to maximize sequential access.

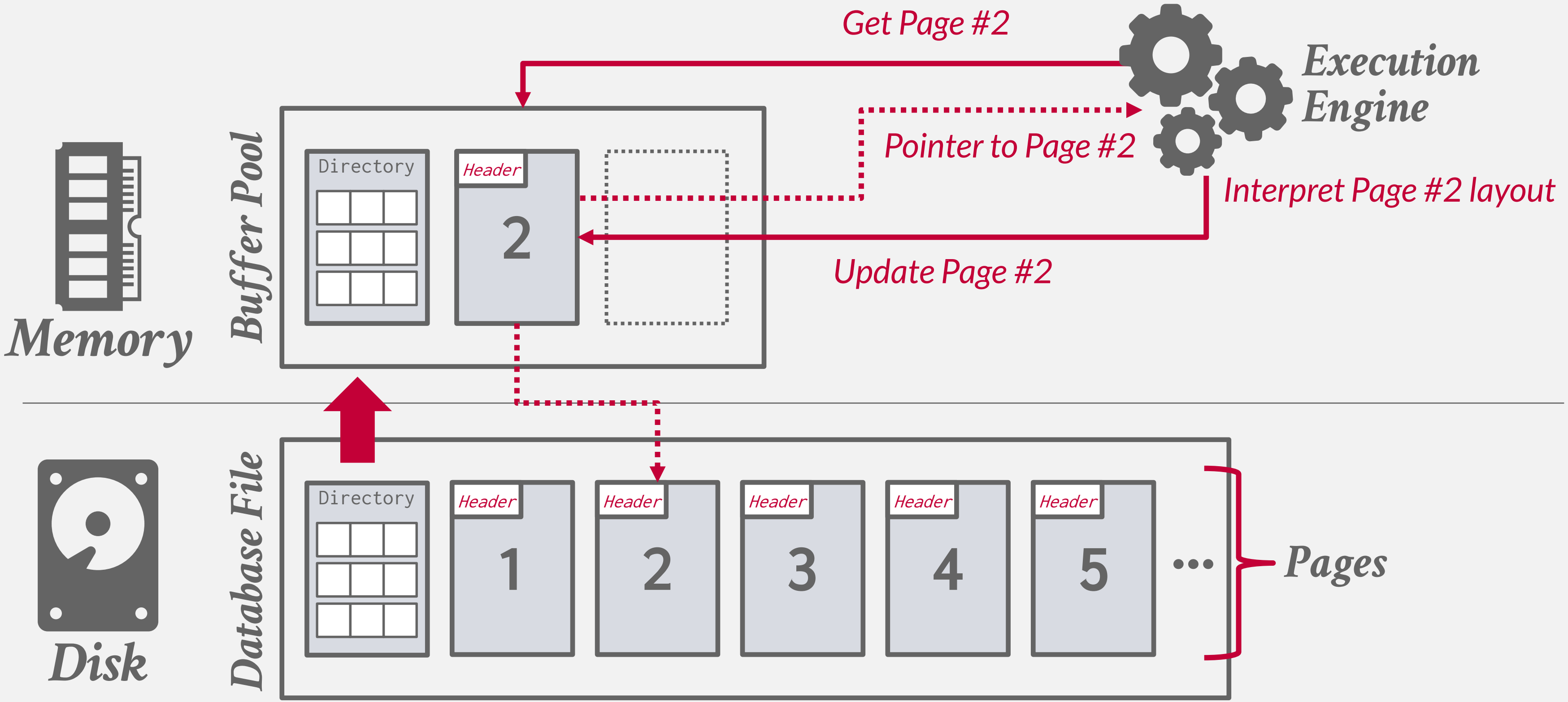
DISK-ORIENTED DBMS



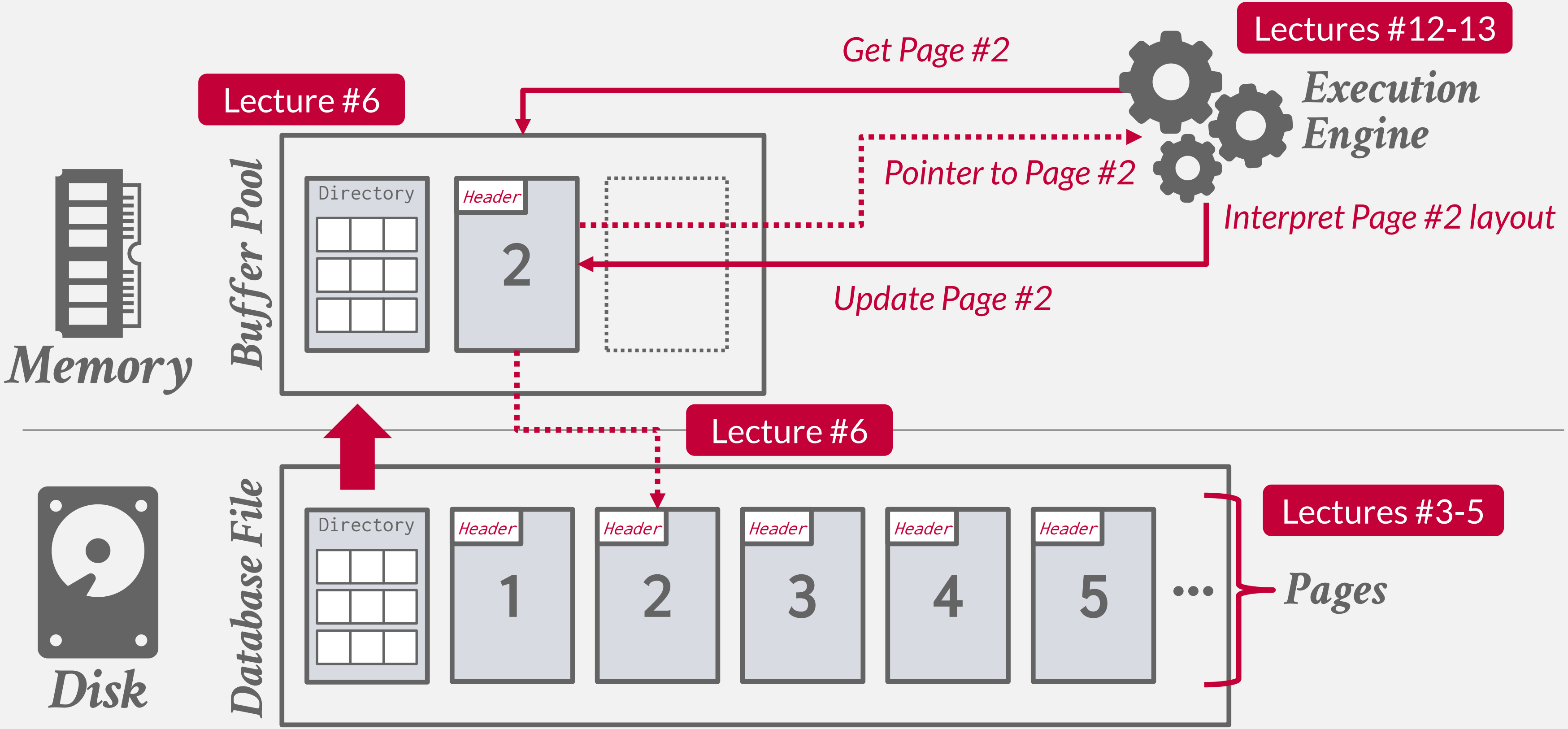
DISK-ORIENTED DBMS



DISK-ORIENTED DBMS



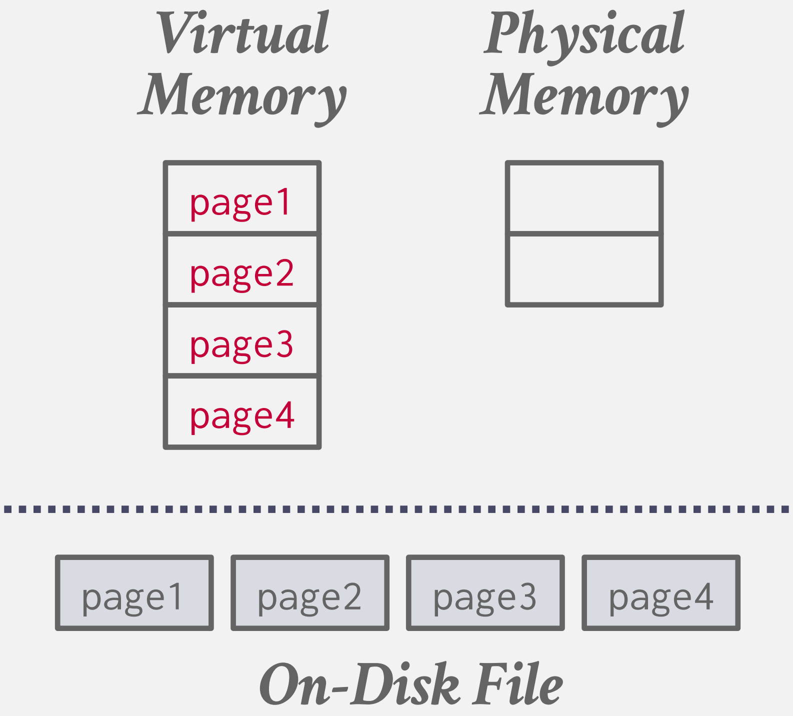
DISK-ORIENTED DBMS



WHY NOT USE THE OS?

The DBMS can use memory mapping (**mmap**) to store the contents of a file into the address space of a program.

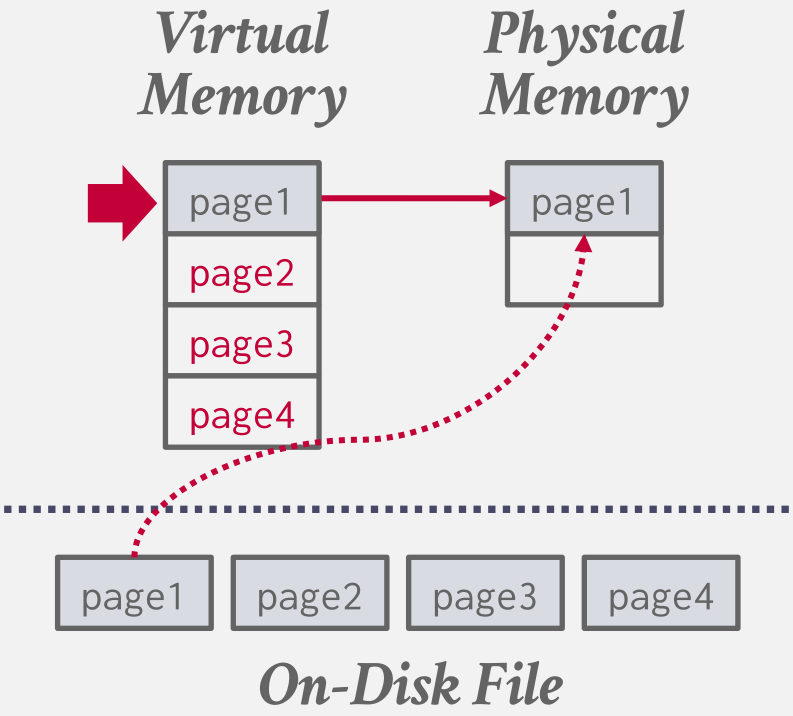
OS is responsible for moving file pages in and out of memory, so the DBMS doesn't need to worry about it.



WHY NOT USE THE OS?

The DBMS can use memory mapping (**mmap**) to store the contents of a file into the address space of a program.

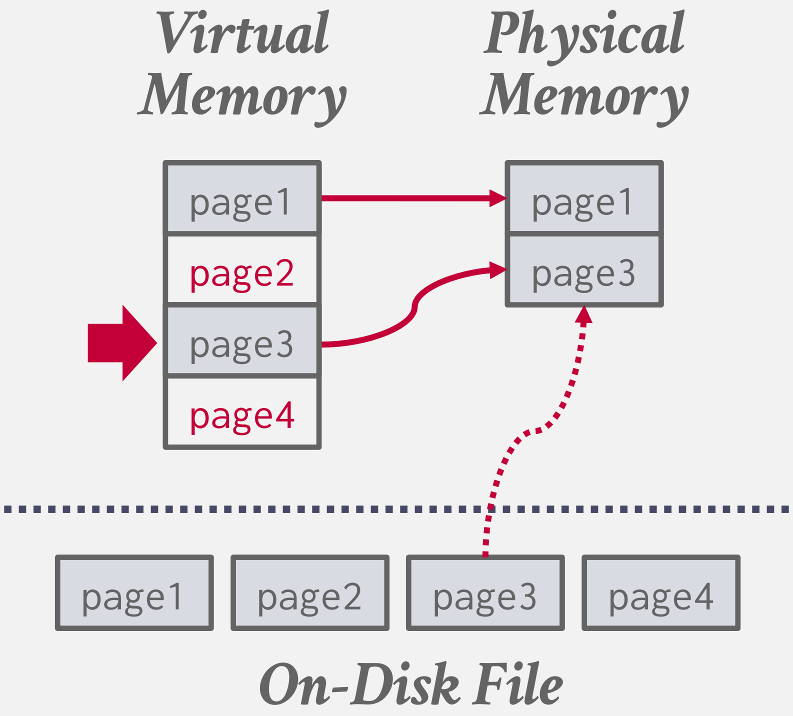
OS is responsible for moving file pages in and out of memory, so the DBMS doesn't need to worry about it.



WHY NOT USE THE OS?

The DBMS can use memory mapping (`mmap`) to store the contents of a file into the address space of a program.

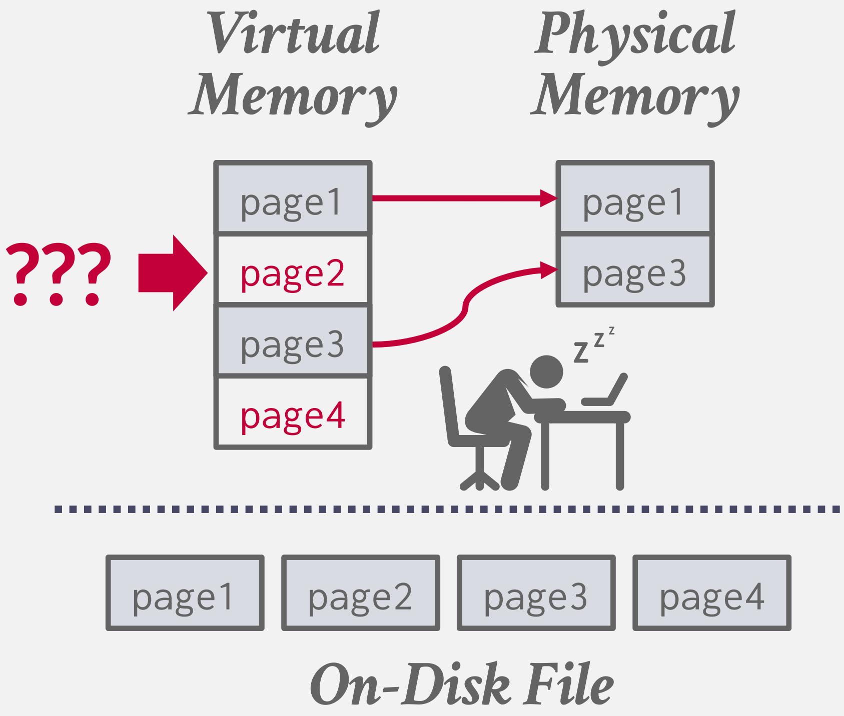
OS is responsible for moving file pages in and out of memory, so the DBMS doesn't need to worry about it.



WHY NOT USE THE OS?

The DBMS can use 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.



WHY NOT USE THE OS?

What if we allow multiple threads to access the **mmap** files to hide page fault stalls?

This works good enough for read-only access.
It is complicated when there are multiple writers…

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. 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, RavenDB, levelDB, elasticsearch, QuestDB

Partial Usage

MongoDB, SingleStore, 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

monetdb
LMDB
RAVENDB
levelDB
elasticsearch
QuestDB

Partial Usage

MongoDB
SingleStore
SQLite
influxdb

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

DBMS (almost) always wants t
itself and can do a better job th

→ Flushing dirty pages to disk in th
→ 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

DATABASE STORAGE

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

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

TODAY'S AGENDA

File Storage
Page Layout
Tuple Layout

FILE STORAGE

The DBMS stores a database as one or more files on disk typically in a proprietary format.

→ The OS doesn't know anything about the contents of these files.
→ We will discuss portable file formats next week…

Early systems in the 1980s used custom filesystems on raw block storage.

→ Some "enterprise" DBMSs still support this.
→ Most newer DBMSs do not do this.

STORAGE MANAGER

The storage manager is responsible for maintaining a database's files.

→ Some do their own scheduling for reads and writes to improve spatial and temporal locality of pages.

It organizes the files as a collection of pages.

→ Tracks data read/written to pages.
→ Tracks the available space.

A DBMS typically does not maintain multiple copies of a page on disk.

→ Assume this happens above/below storage manager.

DATABASE PAGES

A page is a fixed-size block of data.
- → It can contain tuples, meta-data, indexes, log records…
- → Most systems do not mix page types.
- → Some systems require a page to be self-contained.

Each page is given a unique identifier.
- → The DBMS uses an indirection layer to map page IDs to physical locations.

DATABASE PAGES

There are three different notions of "pages" in a DBMS:
→ Hardware Page (usually 4KB)
→ OS Page (usually 4KB, x64 2MB/1GB)
→ Database Page (512B-32KB)

A hardware page is the largest block of data that the storage device can guarantee failsafe writes.

Default DB Page Sizes

4KB: SQLite, ORACLE, IBM DB2, RocksDB, WIREDTIGER

8KB: Microsoft SQL Server, PostgreSQL

16KB: MySQL

PAGE STORAGE ARCHITECTURE

Different DBMSs manage pages in files on disk in different ways.

→ Heap File Organization
→ Tree File Organization
→ Sequential / Sorted File Organization (ISAM)
→ Hashing File Organization

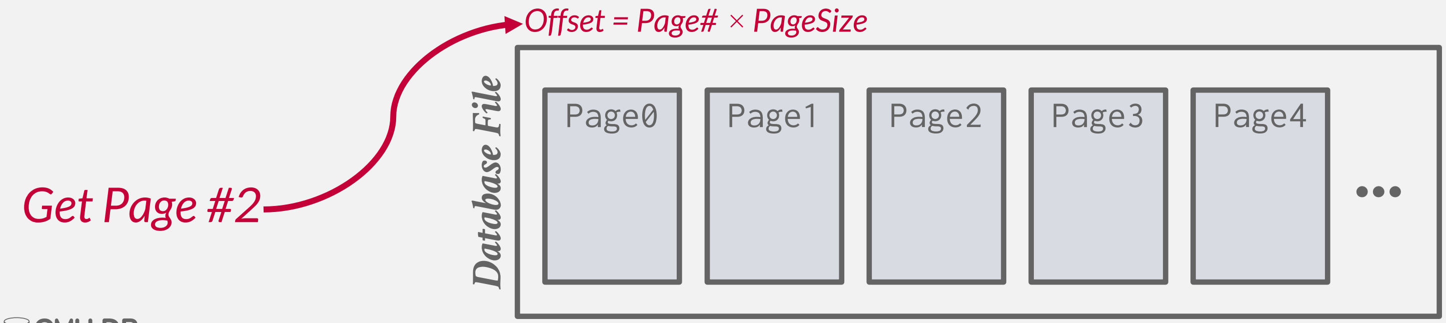
At this point in the hierarchy we don't need to know anything about what is inside of the pages.

HEAP FILE

A heap file is an unordered collection of pages with tuples that are stored in random order.

→ Create / Get / Write / Delete Page
→ Must also support iterating over all pages.

It is easy to find pages if there is only a single file.

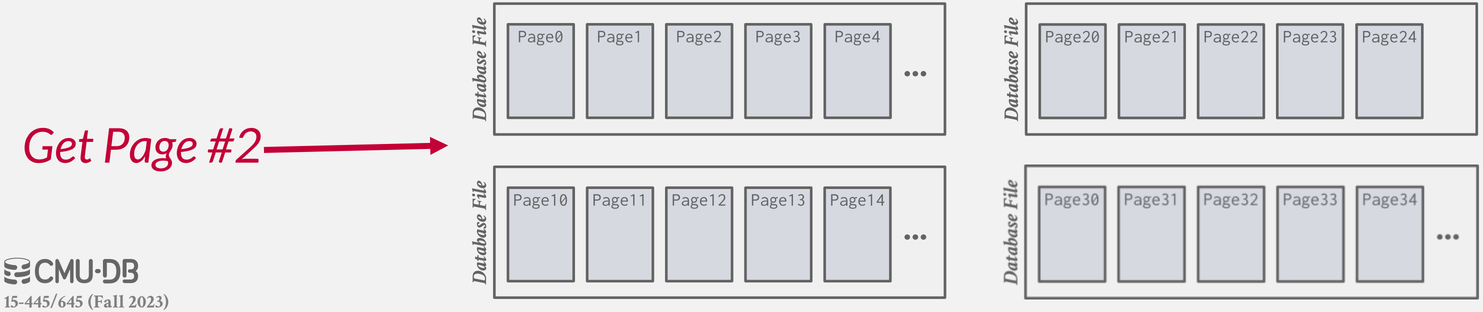


HEAP FILE

A heap file is an unordered collection of pages with tuples that are stored in random order.

- → Create / Get / Write / Delete Page
- → Must also support iterating over all pages.

It is easy to find pages if there is only a single file.



HEAP FILE

A heap file is an unordered collection of pages with tuples that are stored in random order.

→ Create / Get / Write / Delete Page
→ Must also support iterating over all pages.

It is easy to find pages if there is only a single file.

Need meta-data to keep track of what pages exist in multiple files and which ones have free space.

HEAP FILE: PAGE DIRECTORY

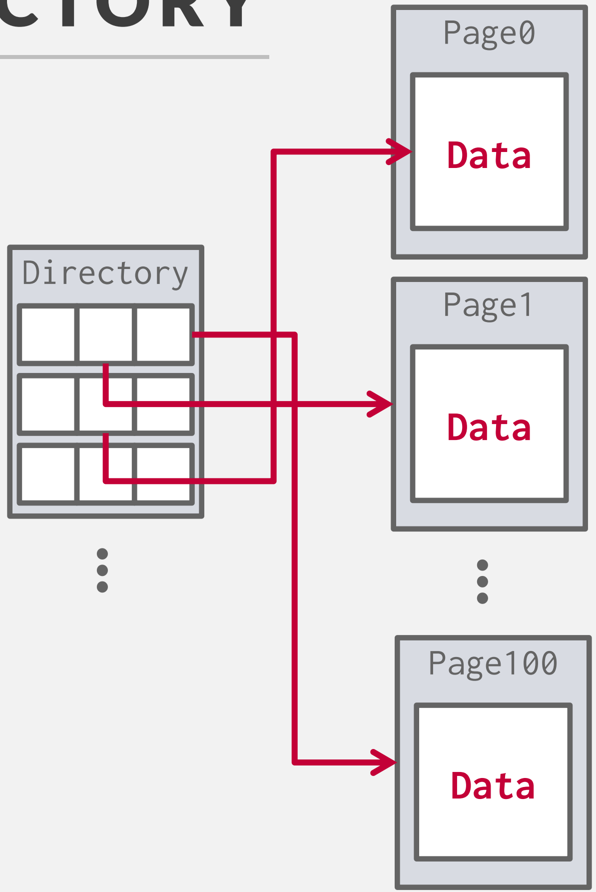
The DBMS maintains special pages that tracks the location of data pages in the database files.

→ Must make sure that the directory pages are in sync with the data pages.

The directory also records meta-data about available space:

→ The number of free slots per page.

→ List of free / empty pages.

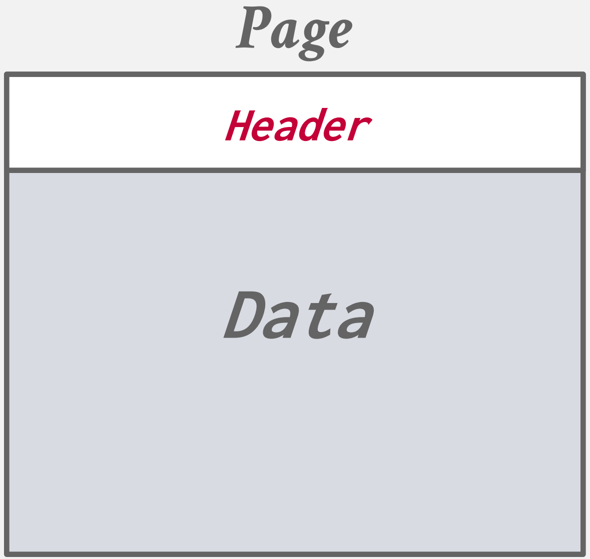


TODAY'S AGENDA

File Storage
Page Layout
Tuple Layout

PAGE HEADER

Every page contains a header of meta-data about the page's contents.
→ Page Size
→ Checksum
→ DBMS Version
→ Transaction Visibility
→ Compression / Encoding Meta-data
→ Schema Information
→ Data Summary / Sketches



Some systems require pages to be self-contained (e.g., Oracle).

PAGE LAYOUT

For any page storage architecture, we now need to decide how to organize the data inside of the page.

→ We are still assuming that we are only storing tuples in a row-oriented storage model. Lecture #5

Approach #1: Tuple-oriented Storage ← Today

Approach #2: Log-structured Storage

Approach #3: Index-organized Storage

PAGE LAYOUT

For any page storage architecture, we now need to decide how to organize the data inside of the page.

→ We are still assuming that we are only storing tuples in a row-oriented storage model.

Lecture #5

Approach #1: Tuple-oriented Storage

Approach #2: Log-structured Storage

Approach #3: Index-organized Storage

Lecture #4

TUPLE-ORIENTED STORAGE

How to store tuples in a page?

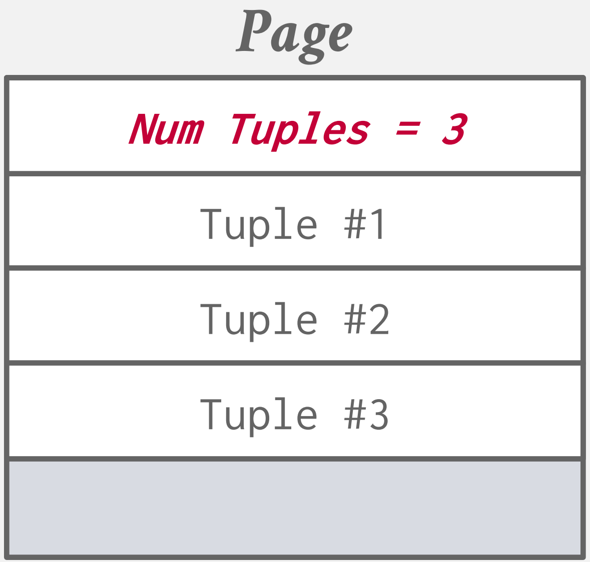
Strawman Idea: Keep track of the number of tuples in a page and then just append a new tuple to the end.

Page

TUPLE-ORIENTED STORAGE

How to store tuples in a page?

Strawman Idea: Keep track of the number of tuples in a page and then just append a new tuple to the end.

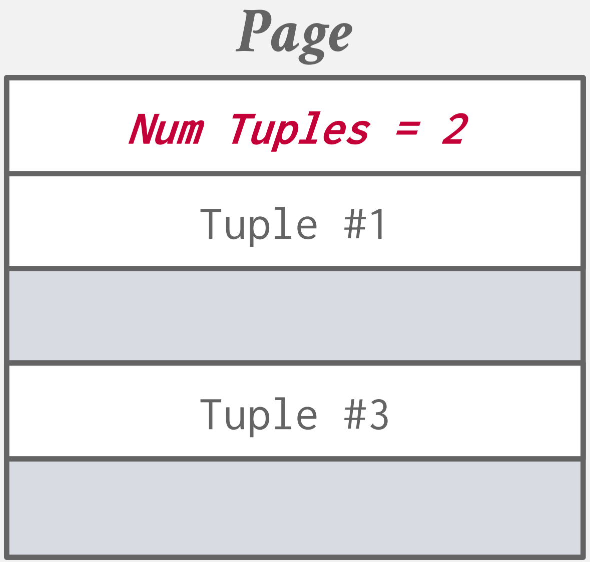


TUPLE-ORIENTED STORAGE

How to store tuples in a page?

Strawman Idea: Keep track of the number of tuples in a page and then just append a new tuple to the end.

→ What happens if we delete a tuple?

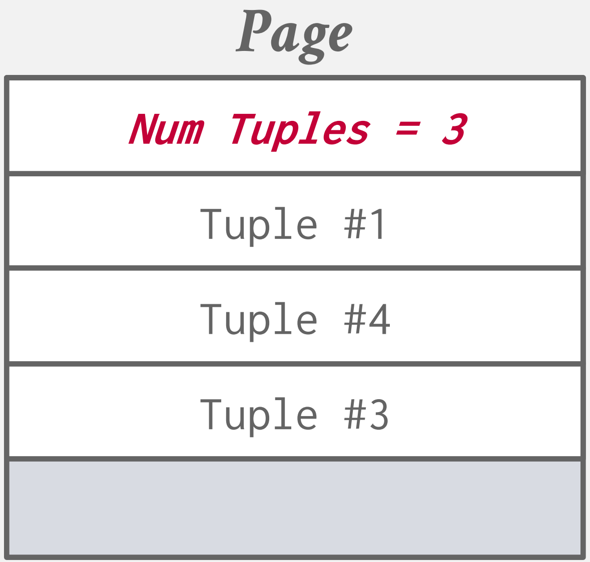


TUPLE-ORIENTED STORAGE

How to store tuples in a page?

Strawman Idea: Keep track of the number of tuples in a page and then just append a new tuple to the end.

→ What happens if we delete a tuple?

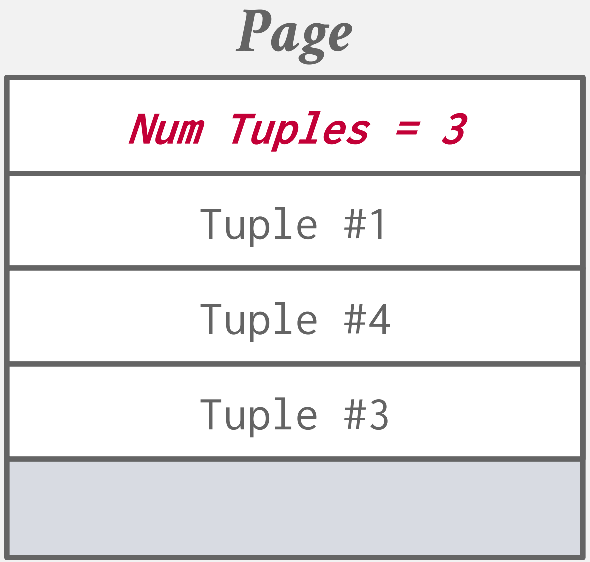


TUPLE-ORIENTED STORAGE

How to store tuples in a page?

Strawman Idea: Keep track of the number of tuples in a page and then just append a new tuple to the end.

→ What happens if we delete a tuple?
→ What happens if we have a variable-length attribute?

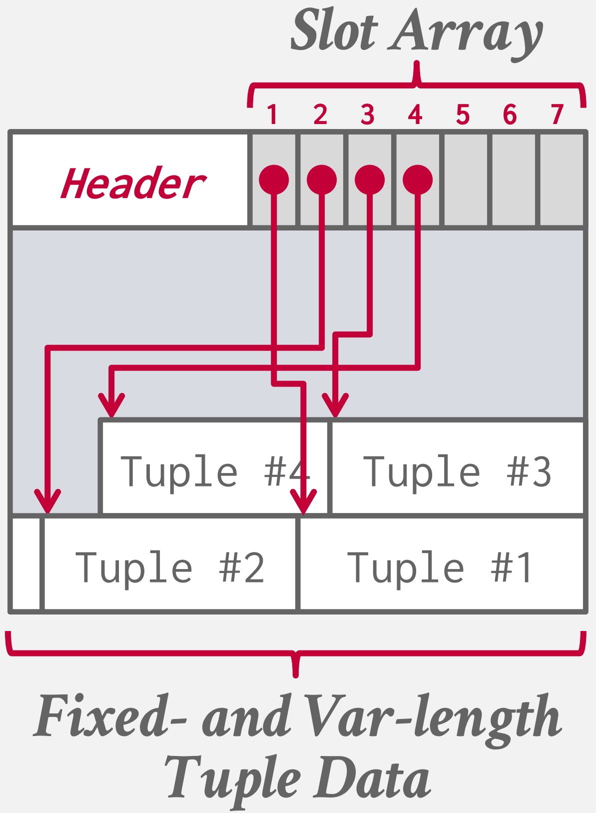


SLOTTED PAGES

The most common layout scheme is called slotted pages.

The slot array maps "slots" to the tuples' starting position offsets.

The header keeps track of:
→ The # of used slots
→ The offset of the starting location of the last slot used.

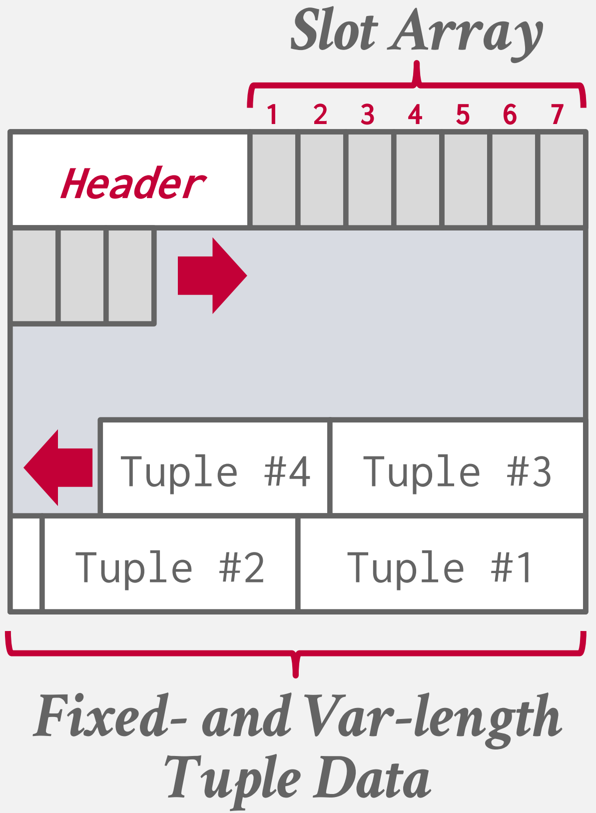


SLOTTED PAGES

The most common layout scheme is called slotted pages.

The slot array maps "slots" to the tuples' starting position offsets.

The header keeps track of:
→ The # of used slots
→ The offset of the starting location of the last slot used.

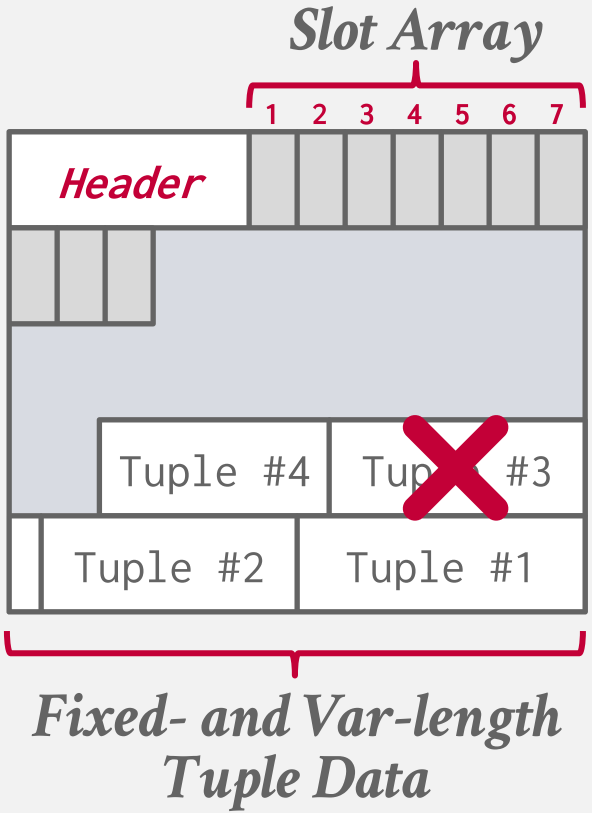


SLOTTED PAGES

The most common layout scheme is called slotted pages.

The slot array maps "slots" to the tuples' starting position offsets.

The header keeps track of:
→ The # of used slots
→ The offset of the starting location of the last slot used.

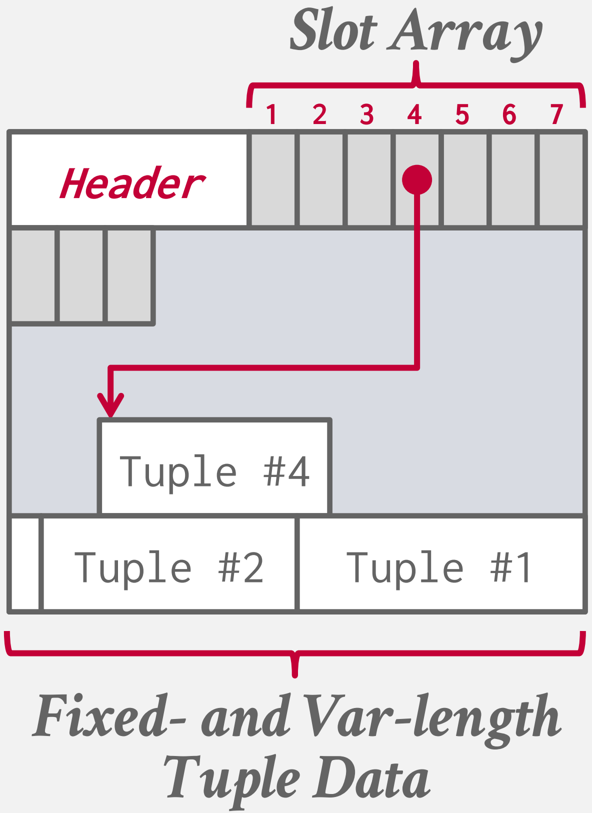


SLOTTED PAGES

The most common layout scheme is called slotted pages.

The slot array maps "slots" to the tuples' starting position offsets.

The header keeps track of:
→ The # of used slots
→ The offset of the starting location of the last slot used.

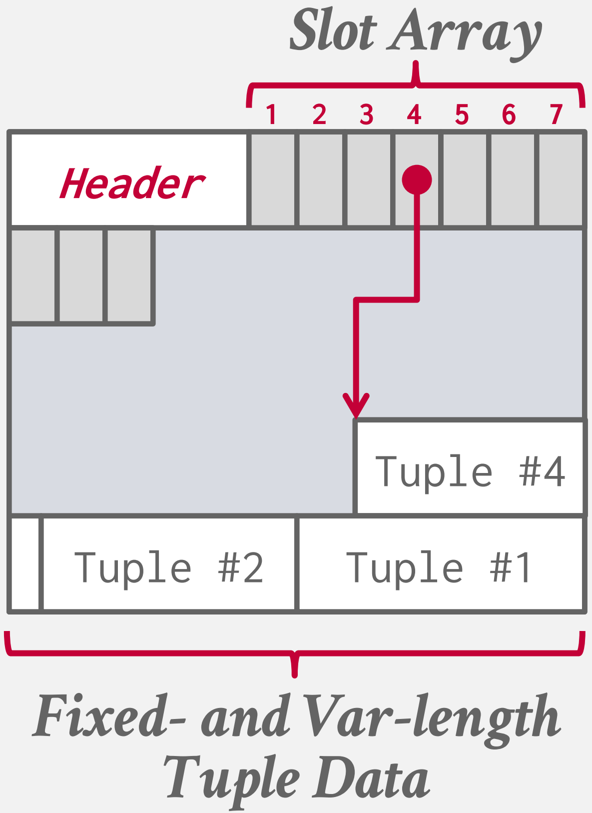


SLOTTED PAGES

The most common layout scheme is called slotted pages.

The slot array maps "slots" to the tuples' starting position offsets.

The header keeps track of:
→ The # of used slots
→ The offset of the starting location of the last slot used.



RECORD IDS

The DBMS assigns each logical tuple a unique record identifier that represents its physical location in the database.

→ File Id, Page Id, Slot #
→ Most DBMSs do not store ids in tuple.
→ SQLite uses ROWID as the true primary key and stores them as a hidden attribute.

Applications should never rely on these IDs to mean anything.

PostgreSQL
CTID (6-bytes)

SQLite
ROWID (8-bytes)

Microsoft SQL Server
%%physloc%% (8-bytes)

ORACLE
ROWID (10-bytes)

TODAY'S AGENDA

File Storage
Page Layout
Tuple Layout

TUPLE LAYOUT

A tuple is essentially a sequence of bytes.

It's the job of the DBMS to interpret those bytes into attribute types and values.

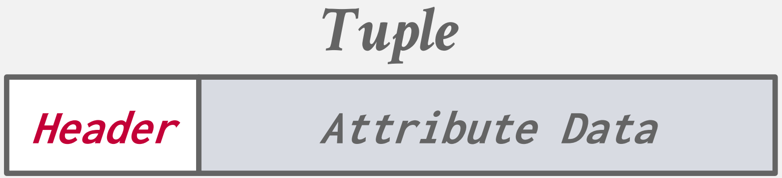
TUPLE HEADER

Each tuple is prefixed with a header that contains meta-data about it.

→ Visibility info (concurrency control)

→ Bit Map for **NULL** values.

We do not need to store meta-data about the schema.

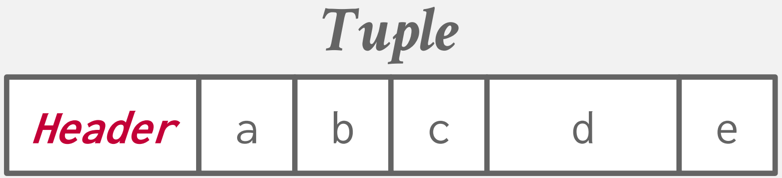


TUPLE DATA

Attributes are typically stored in the order that you specify them when you create the table.

This is done for software engineering reasons (i.e., simplicity).

However, it might be more efficient to lay them out differently.




```sql
CREATE TABLE foo (
  a INT PRIMARY KEY,
  b INT NOT NULL,
  c INT,
  d DOUBLE,
  e FLOAT
);
```

DENORMALIZED TUPLE DATA

DBMS can physically ***denormalize*** (e.g., "pre-join") related tuples and store them together in the same page.

→ Potentially reduces the amount of I/O for common workload patterns.

→ Can make updates more expensive.

```sql
CREATE TABLE foo (
  a INT PRIMARY KEY,
  b INT NOT NULL,
);
```

```sql
CREATE TABLE bar (
  c INT PRIMARY KEY,
  a INT
  ⤷REFERENCES foo (a),
);
```

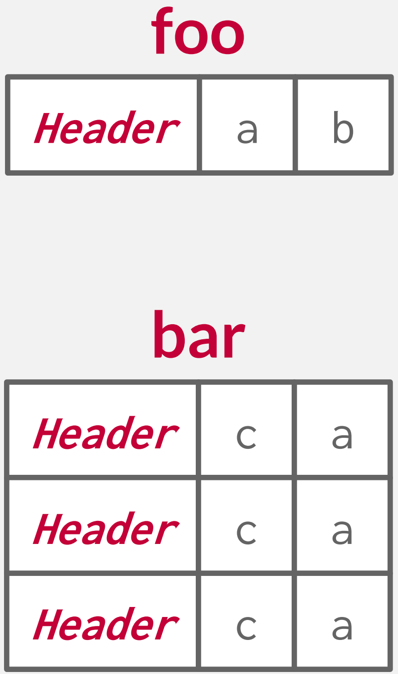
DENORMALIZED TUPLE DATA

DBMS can physically ***denormalize*** (e.g., "pre-join") related tuples and store them together in the same page.

→ Potentially reduces the amount of I/O for common workload patterns.

→ Can make updates more expensive.

```
SELECT * FROM foo JOIN bar
     ON foo.a = bar.a;
```

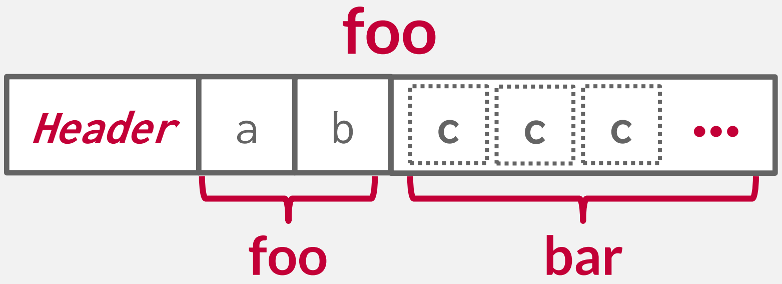


DENORMALIZED TUPLE DATA

DBMS can physically ***denormalize*** (e.g., "pre-join") related tuples and store them together in the same page.

→ Potentially reduces the amount of I/O for common workload patterns.

→ Can make updates more expensive.



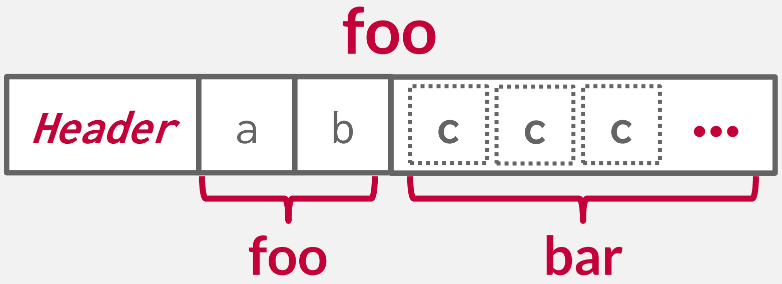

```
SELECT * FROM foo JOIN bar
    ON foo.a = bar.a;
```

DENORMALIZED TUPLE DATA

DBMS can physically ***denormalize*** (e.g., "pre-join") related tuples and store them together in the same page.

→ Potentially reduces the amount of I/O for common workload patterns.

→ Can make updates more expensive.



Not a new idea.

→ IBM System R did this in the 1970s.

→ Several NoSQL DBMSs do this without calling it physical denormalization.

CONCLUSION

Database is organized in pages.

Different ways to track pages.

Different ways to store pages.

Different ways to store tuples.

NEXT CLASS

Log-Structured Storage
Index-Organized Storage
Value Representation
Catalogs