香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

SCHOOL OF
DATA SCIENCE
數據科學學院

Ack: Prof. Jignesh Patel @ CMU
Prof. Andy Pavlo @CMU

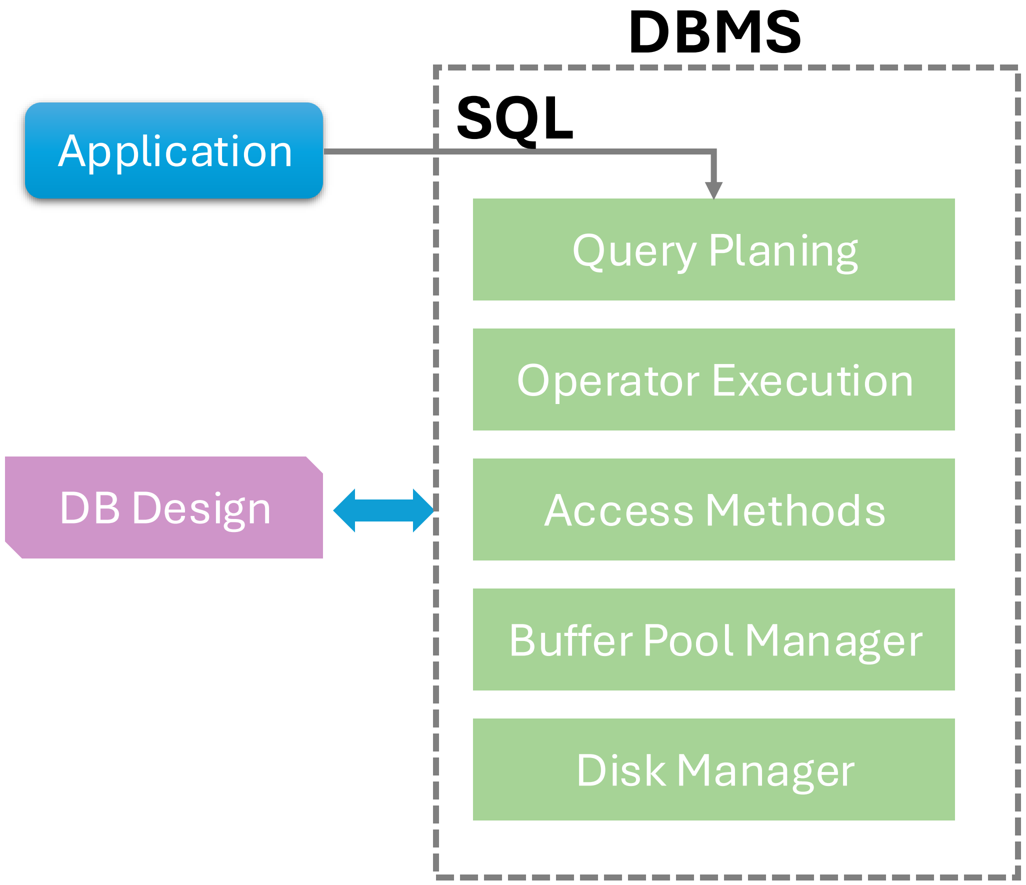
CSC3170
4: Database Storage

Chenhao Ma
School of Data Science
The Chinese University of Hong Kong, Shenzhen

Previous Lectures

- 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).

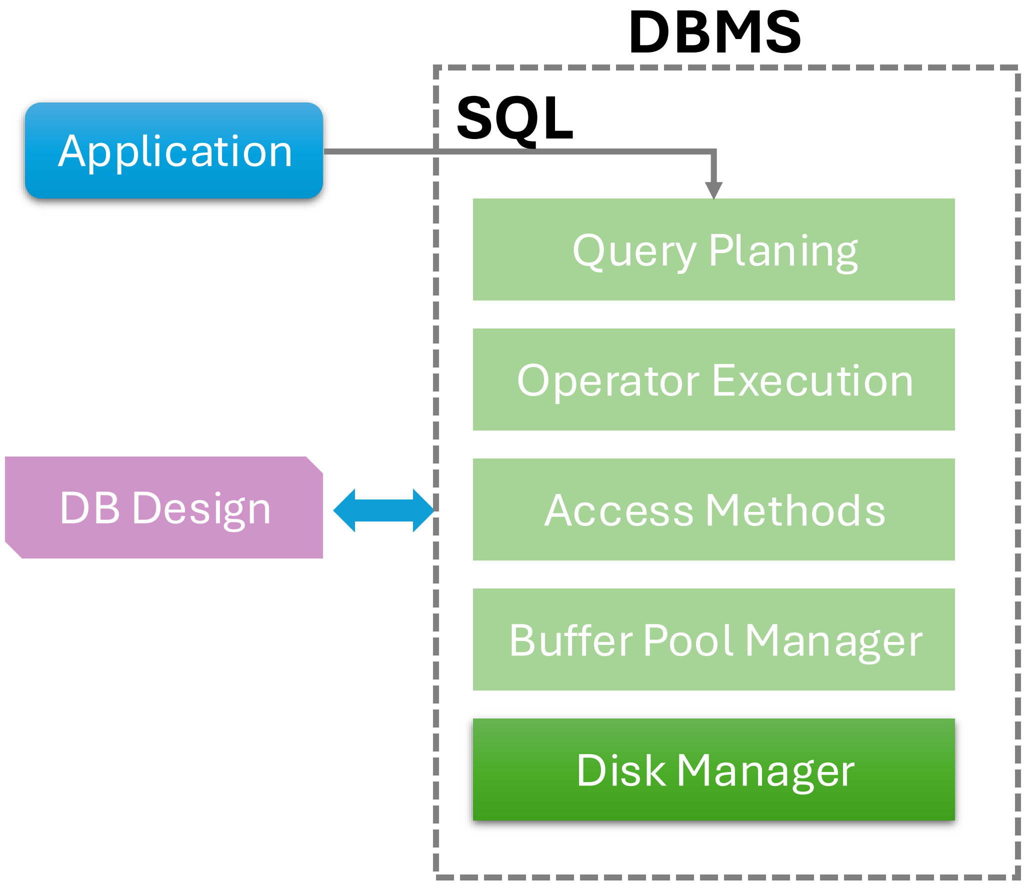
- We will next learn how to build software that manages a database (i.e., a DBMS).



Previous Lectures

- 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).



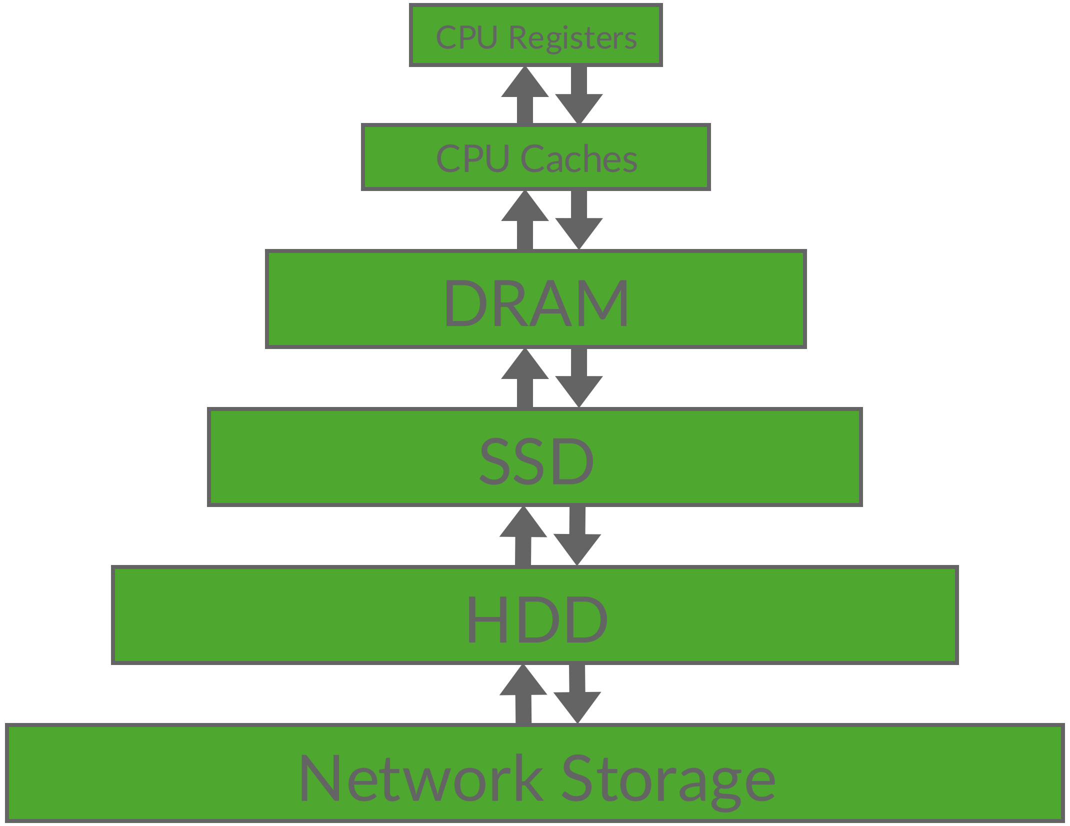
Disk-based Architecture

Disk-based Architecture

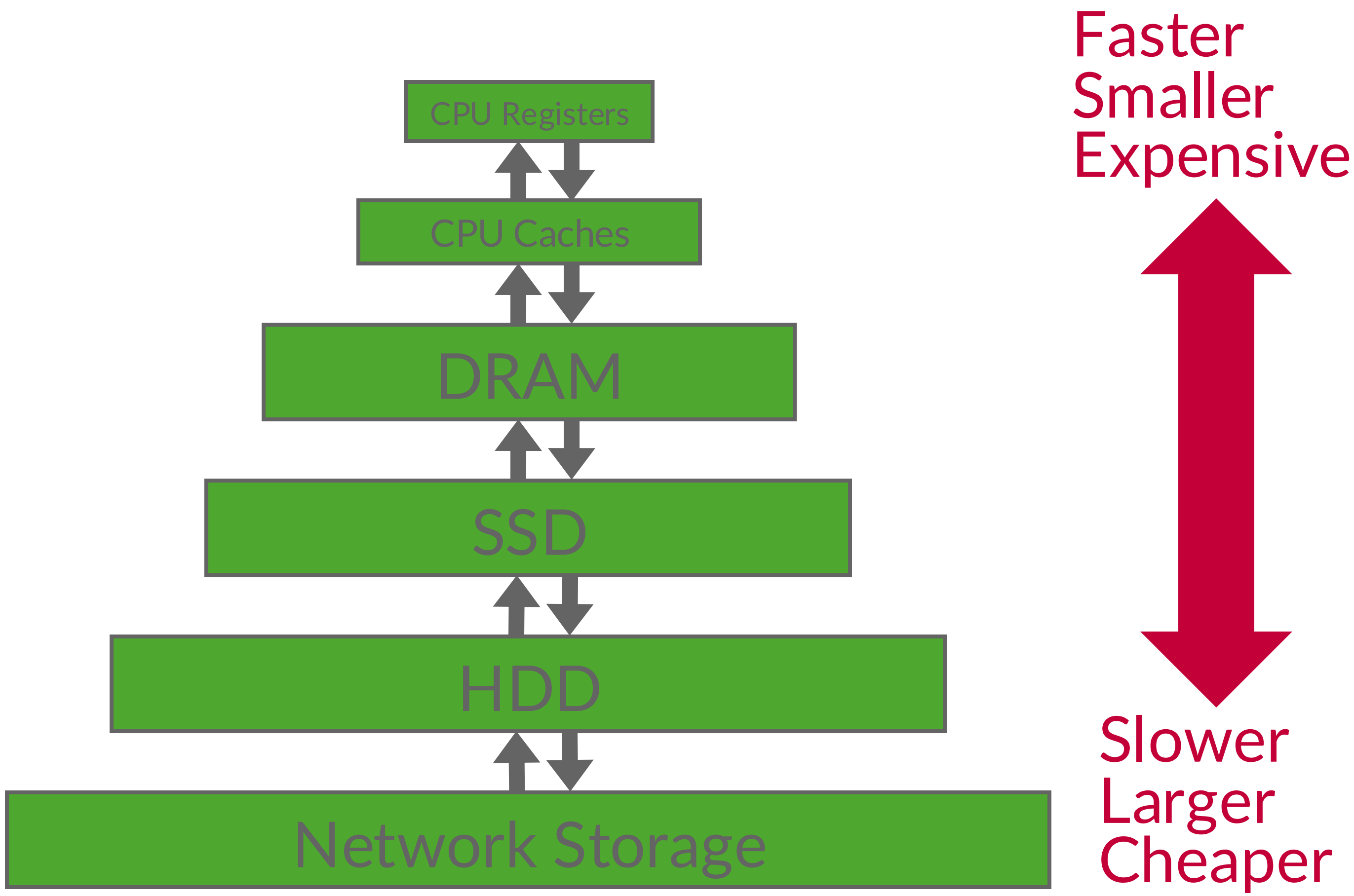
- The DBMS assumes that the primary storage location of the database is on non-volatile **disk**.

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

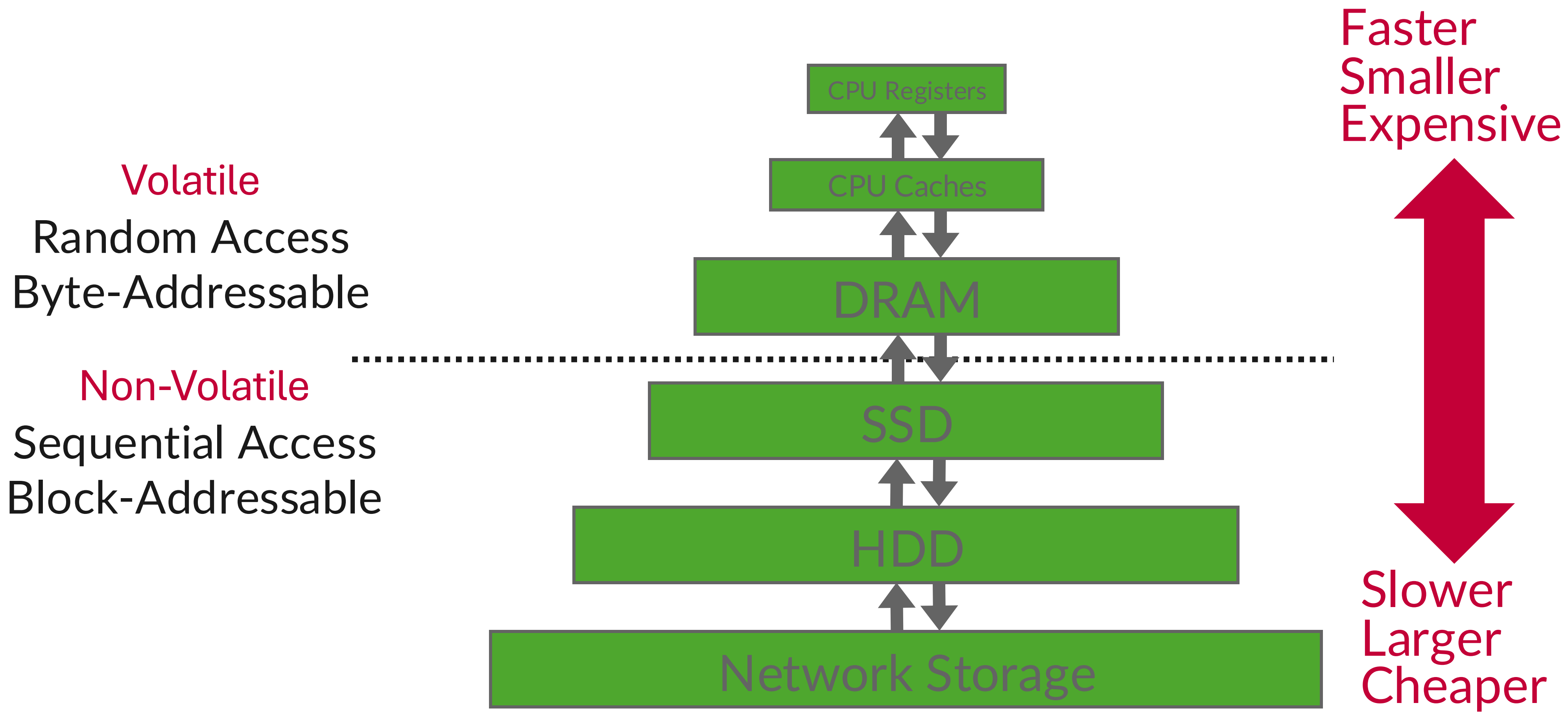
Storage Hierarchy



Storage Hierarchy



Storage Hierarchy



Storage Hierarchy

Volatile

Random Access

Byte-Addressable

Non-Volatile

Sequential Access

Block-Addressable

CPU Registers

CPU Caches

DRAM

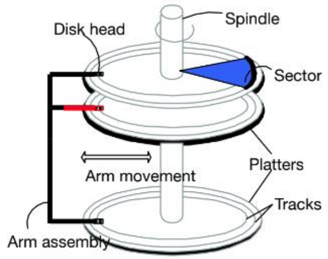
SSD

HDD

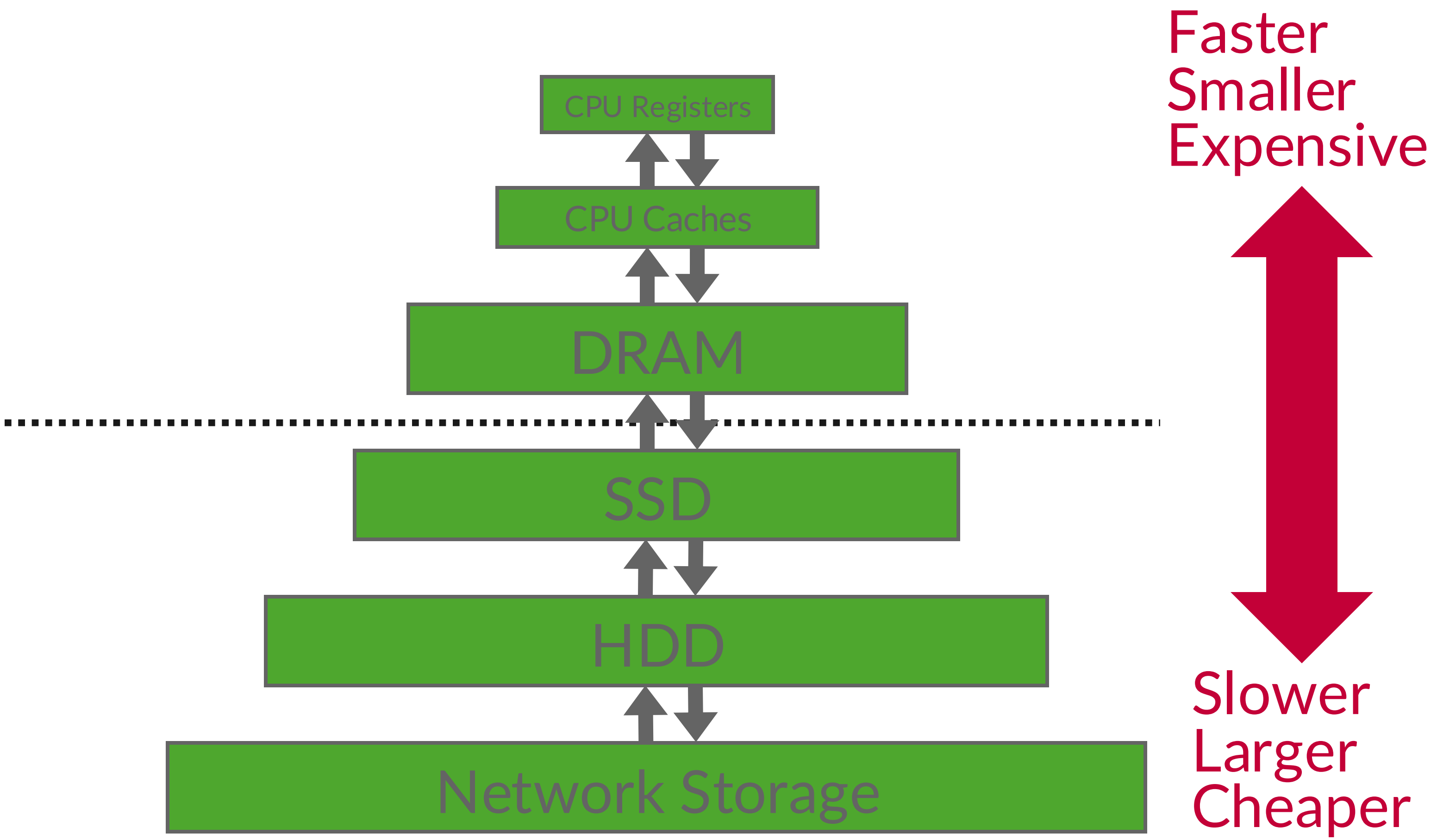
Network Storage

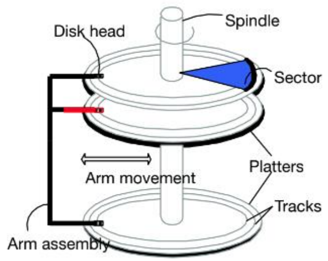
Faster
Smaller
Expensive

Slower
Larger
Cheaper

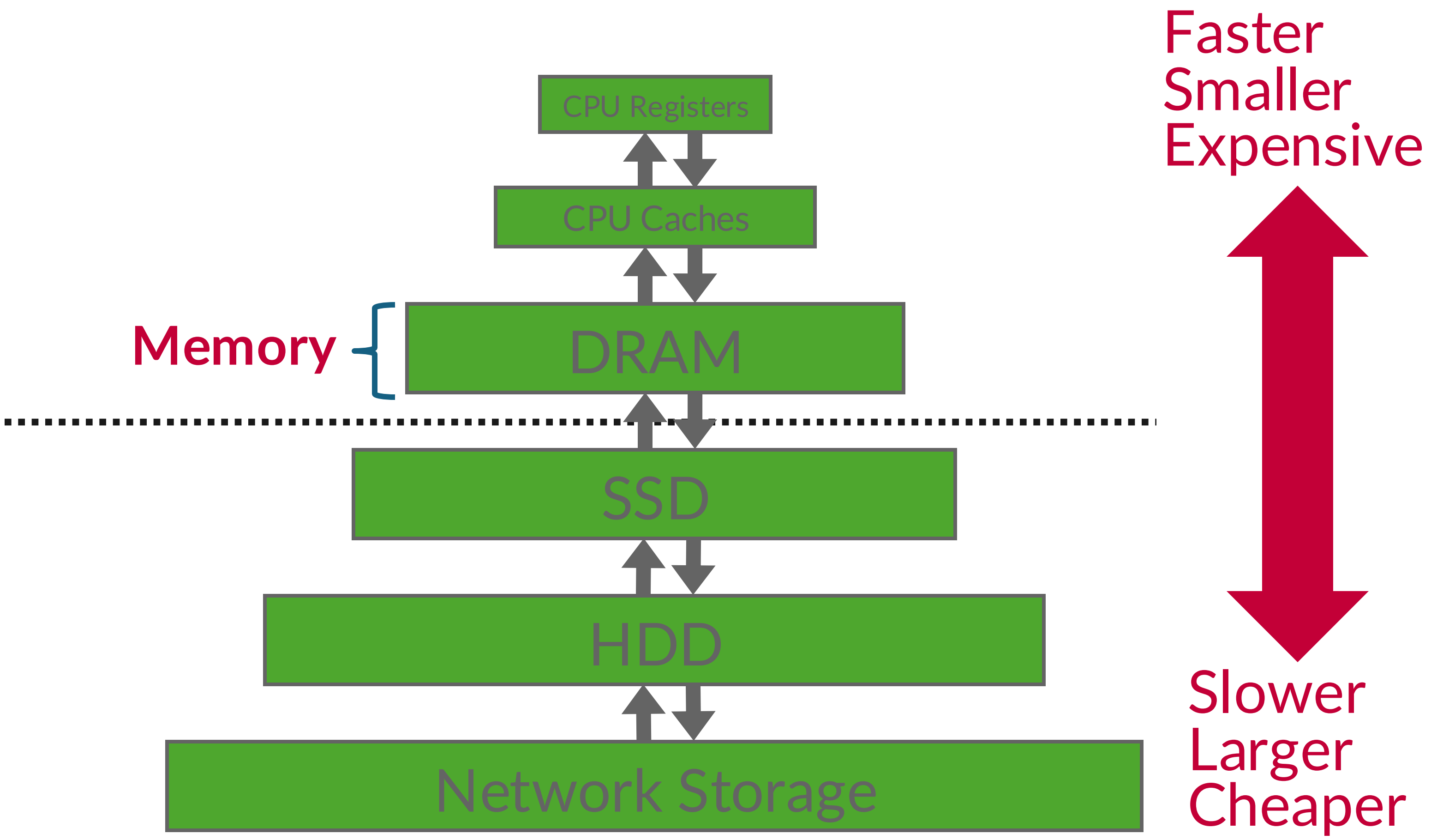


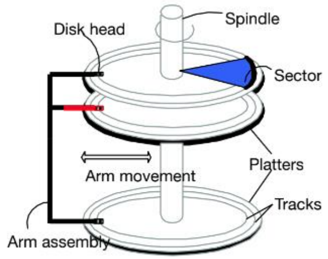
Storage Hierarchy



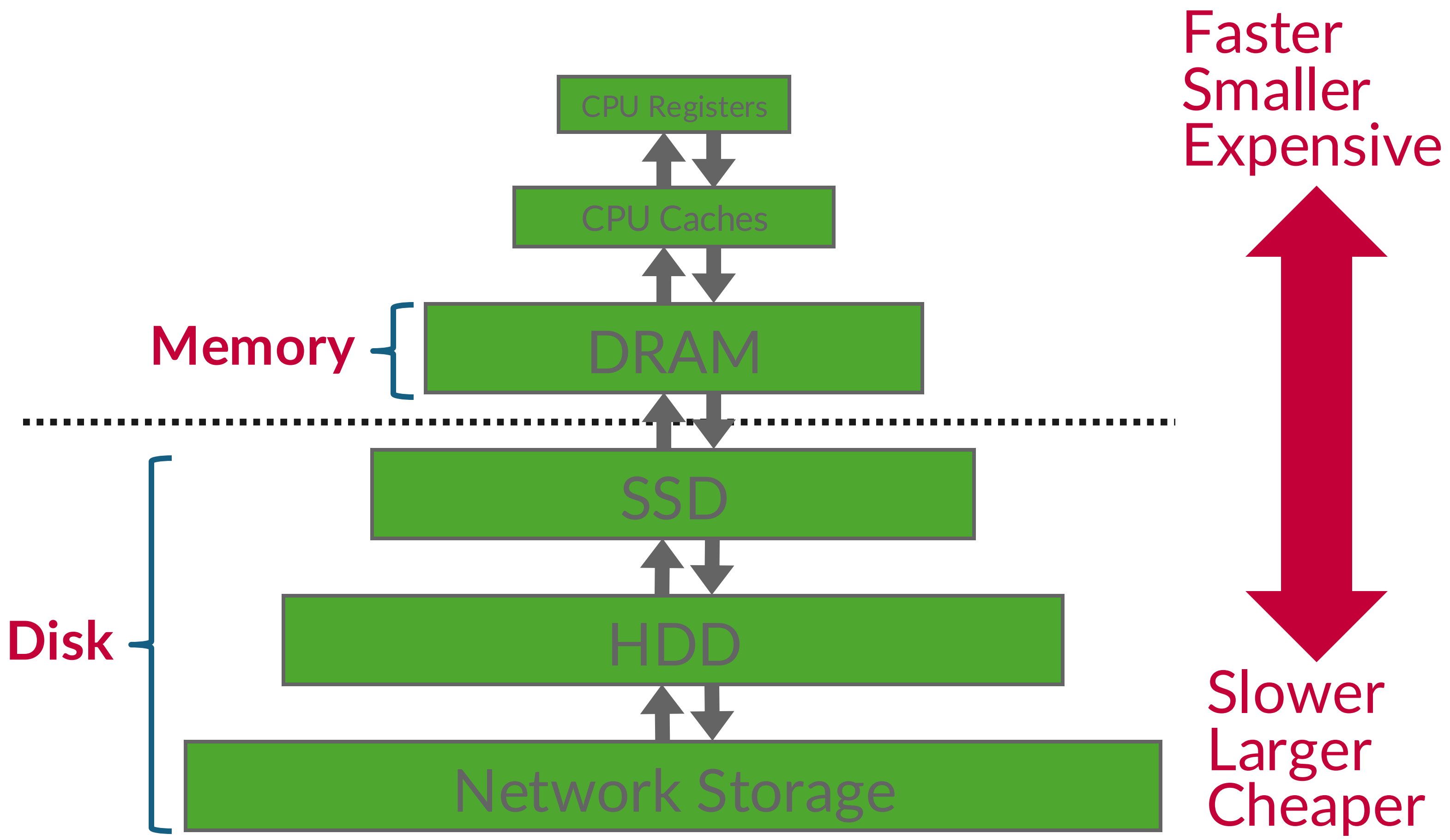


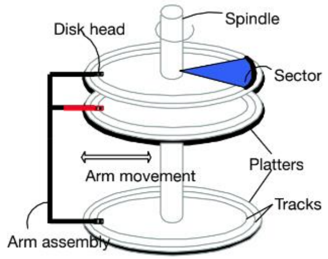
Storage Hierarchy



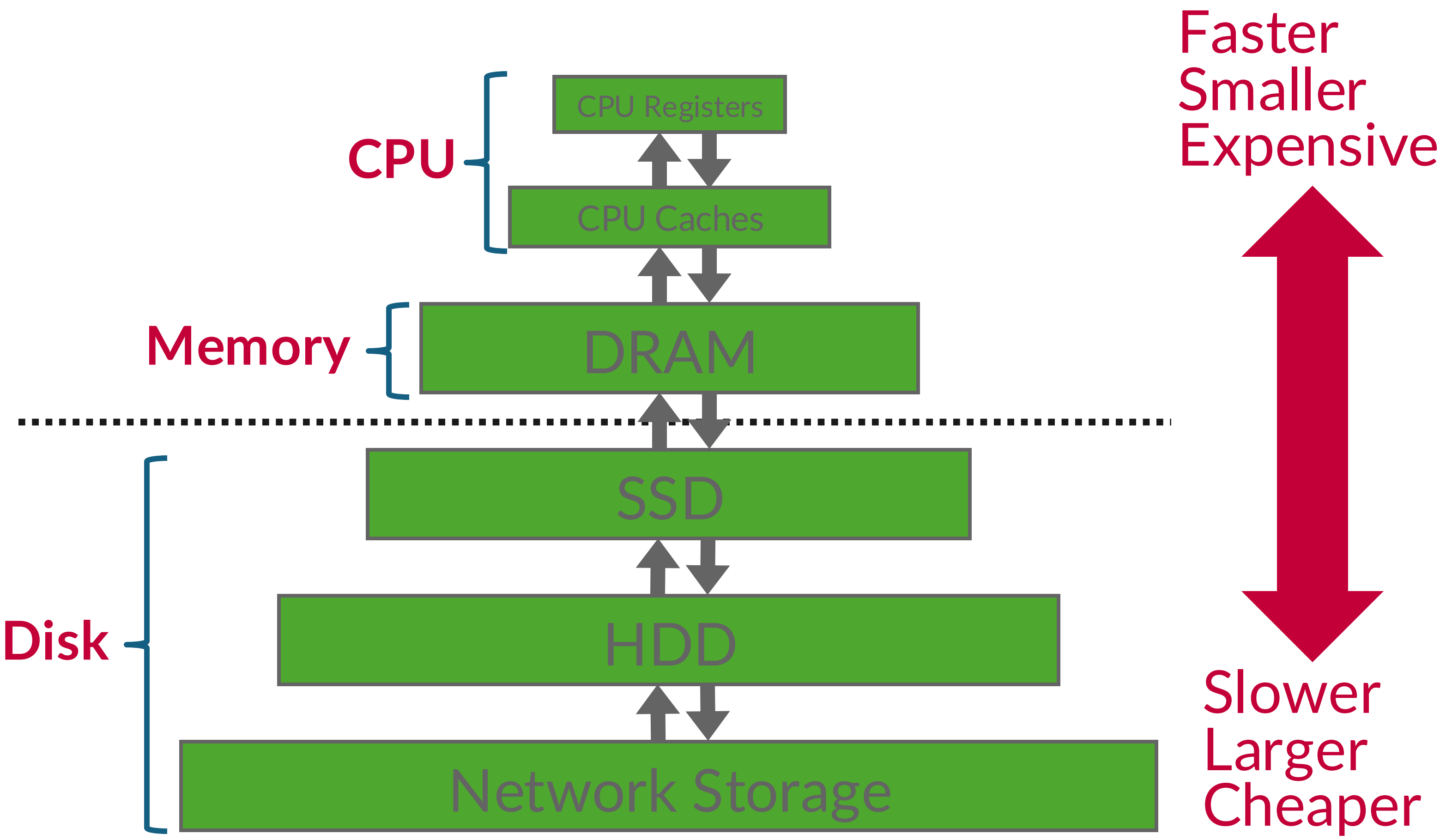


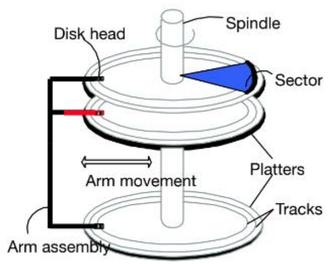
Storage Hierarchy



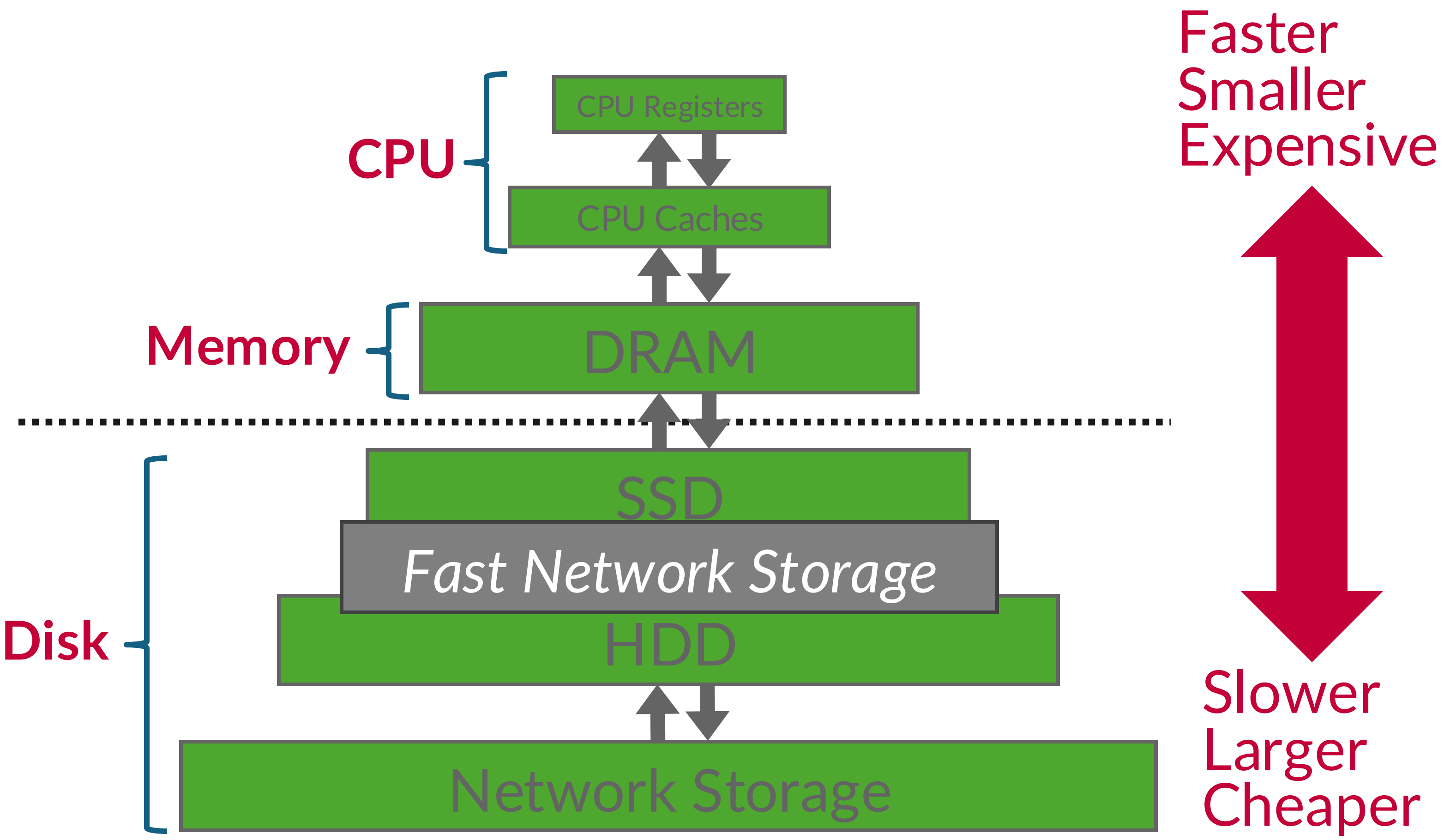


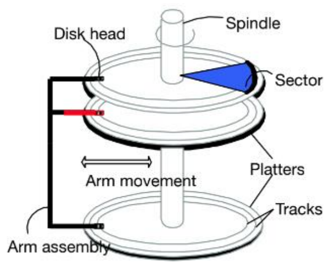
Storage Hierarchy





Storage Hierarchy

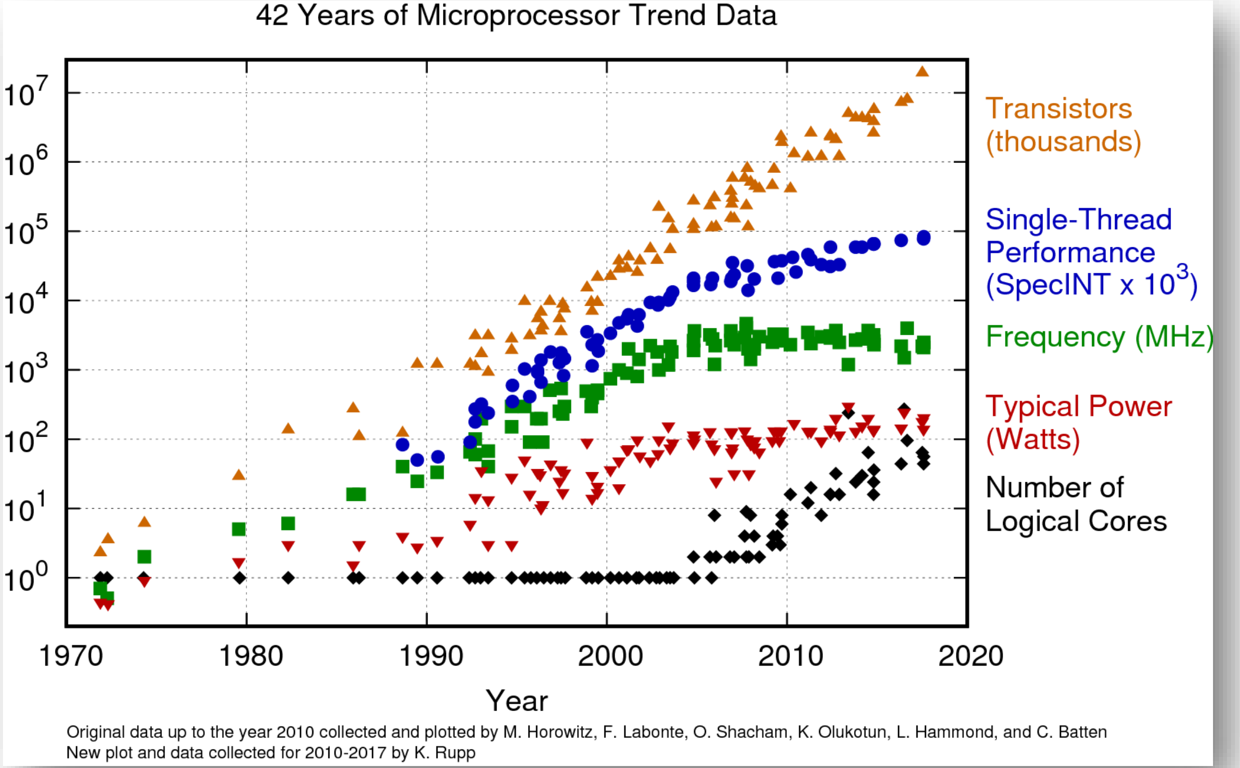




SCHOOL OF DATA SCIENCE

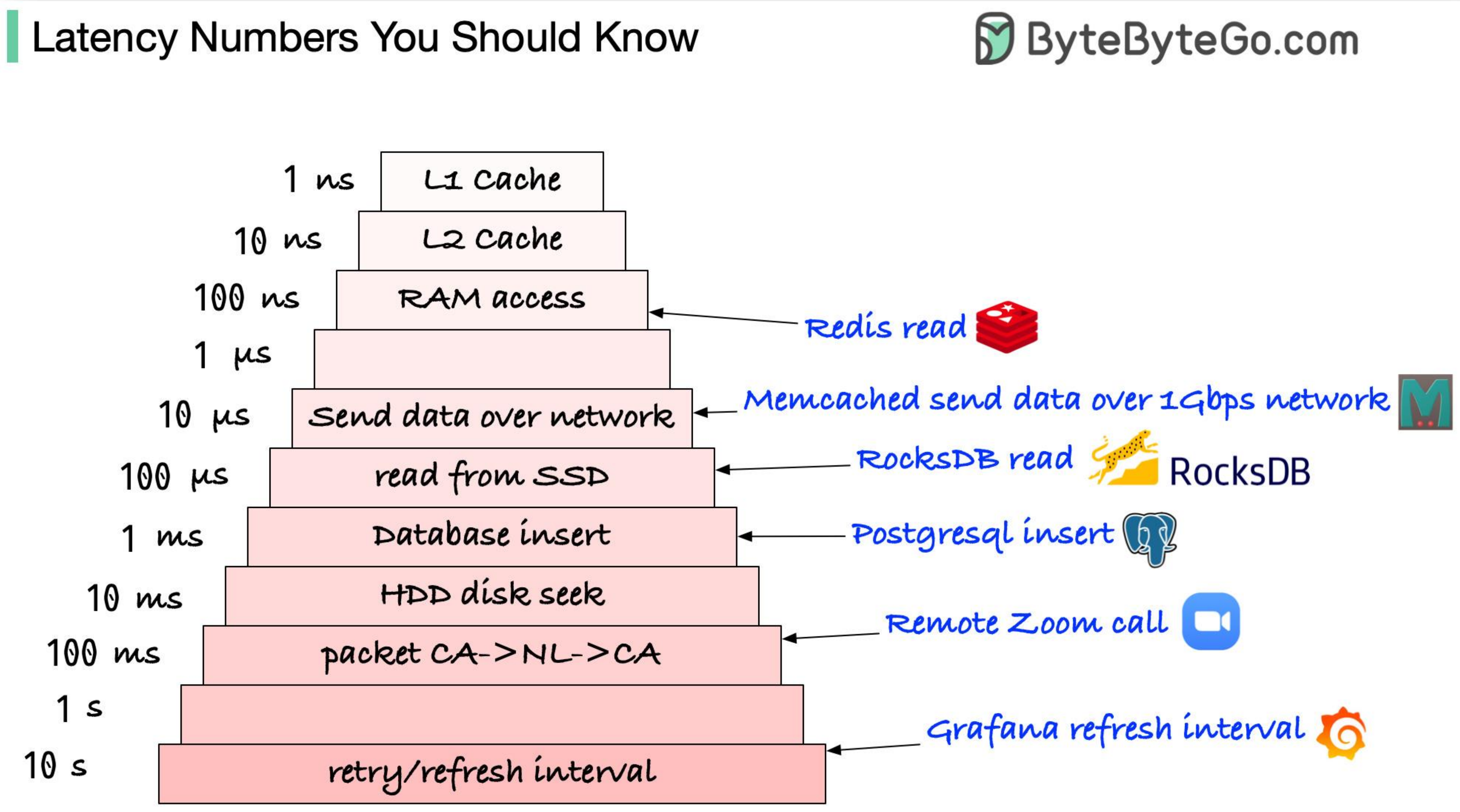
Hardware Trends

- Transistor growth continues.
- The question is how to use this hardware for higher application performance.
- Individual cores are not becoming faster, but there are more cores.
- Every processor is now a “parallel” data machine, and the degree of parallelism is increasing.



https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

SCHOOL OF DATA SCIENCE



https://blog.bytebytego.com/p/ep22-latency-numbers-you-should-know

Every programmer must know these numbers.

Jeff Dean

Sequential v.s. 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 <u>extent</u>.

System Design Goals

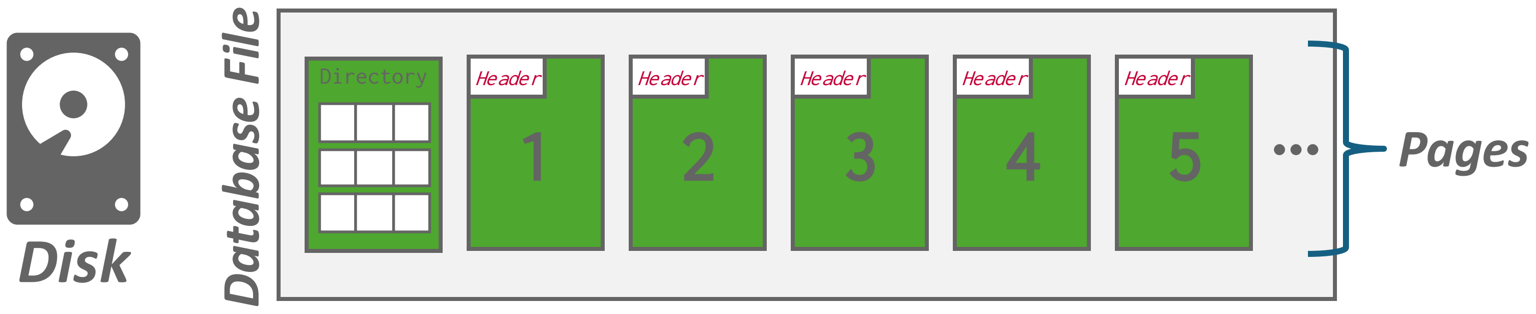
- Allow the DBMS to manage databases that exceed the amount of memory available.

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

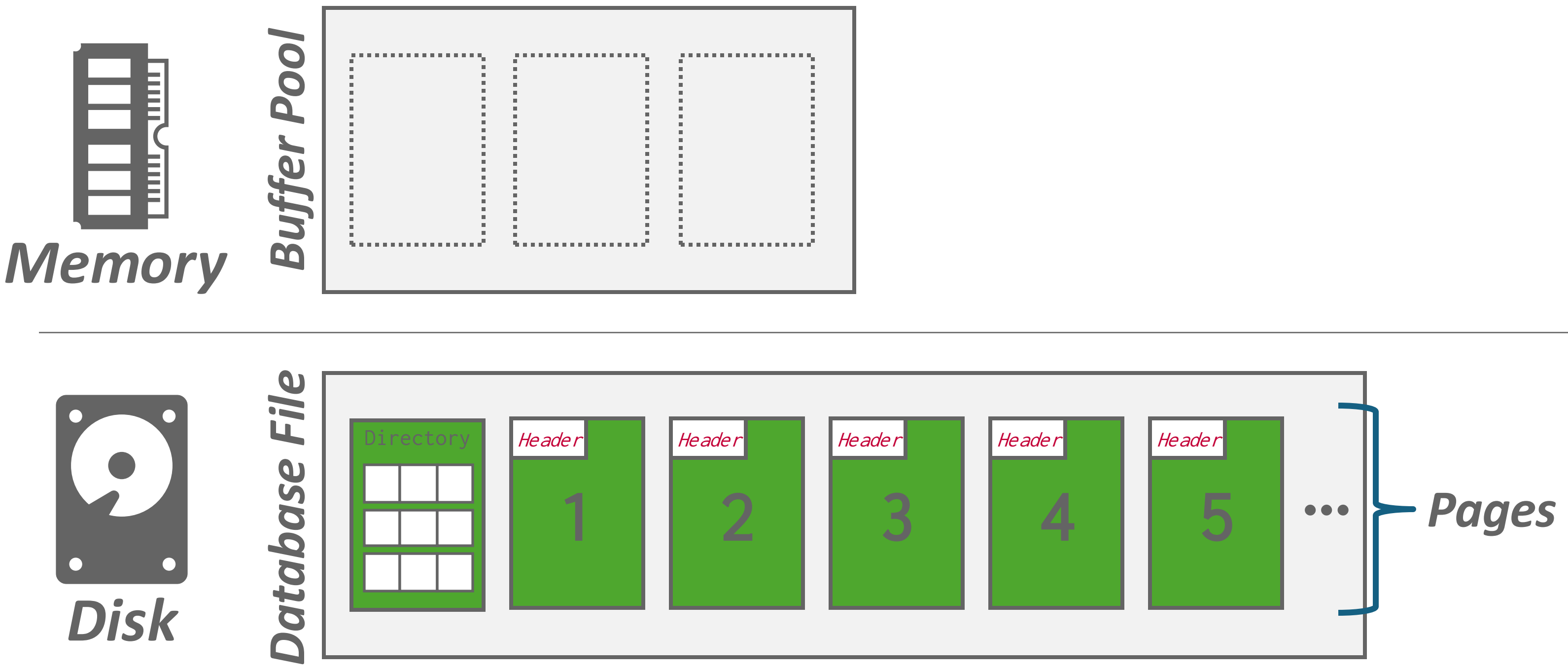
- Random access on disk is usually much slower than sequential access, so the DBMS will want to maximize sequential access.

Disk-oriented DBMS

Disk-oriented DBMS

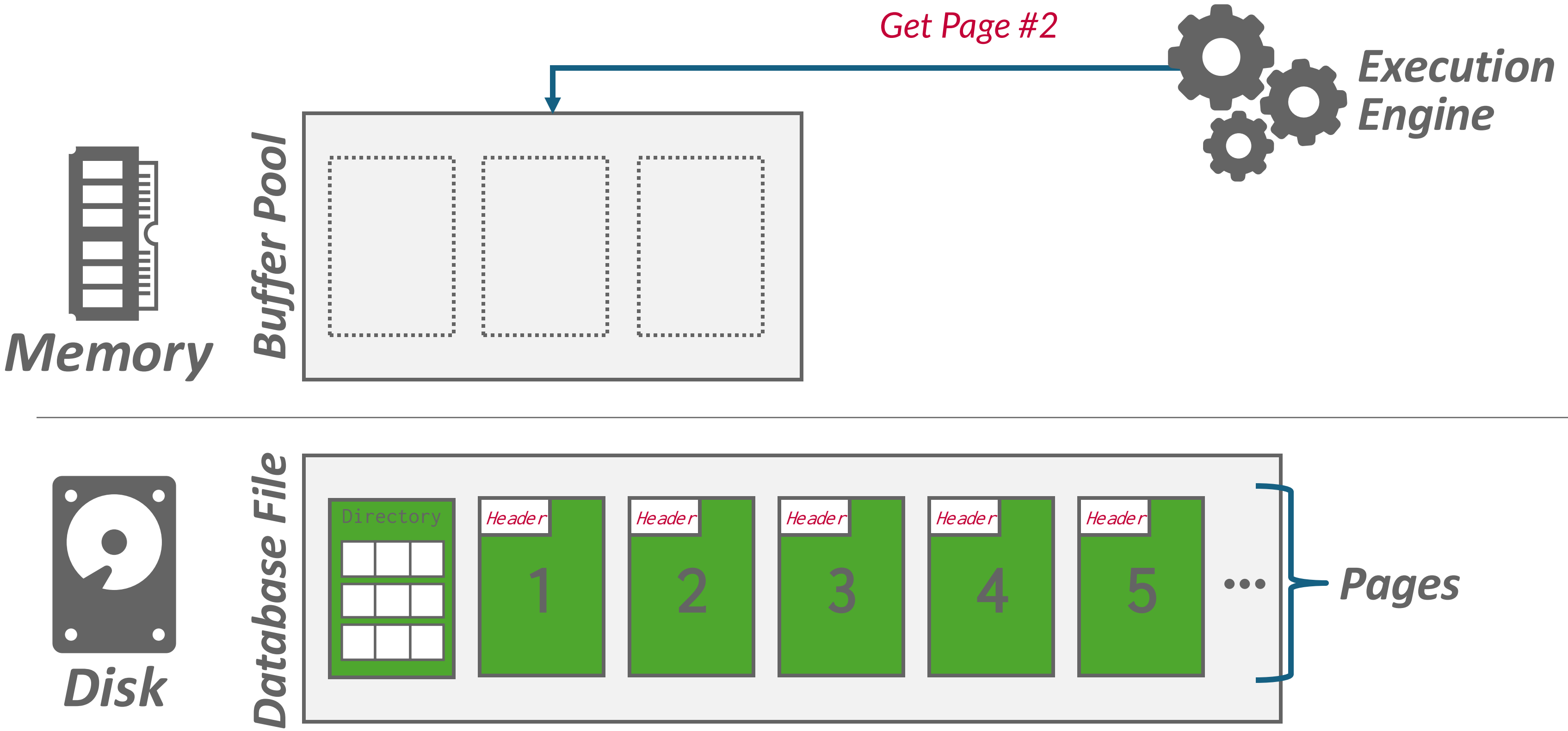


Disk-oriented DBMS

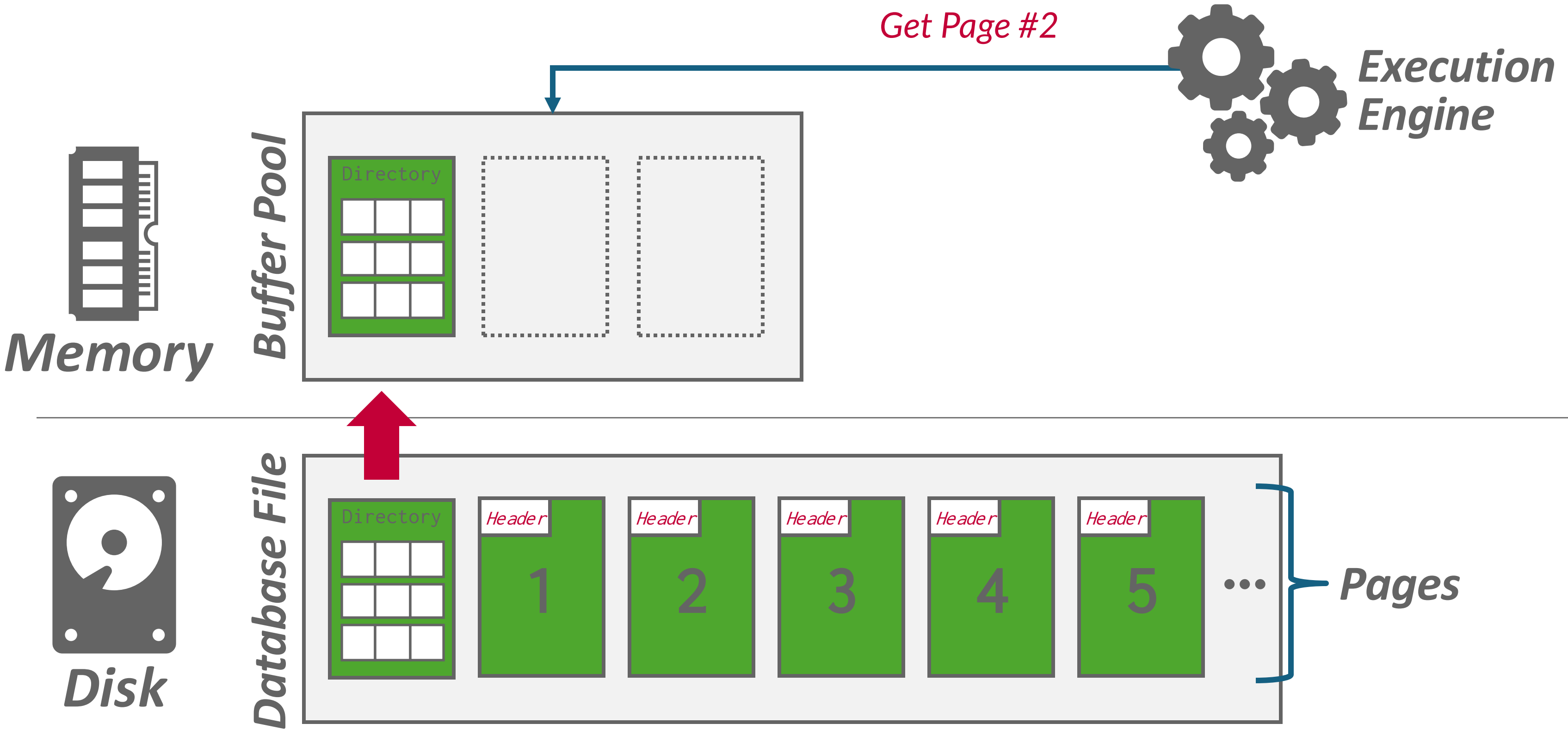


Disk-oriented DBMS

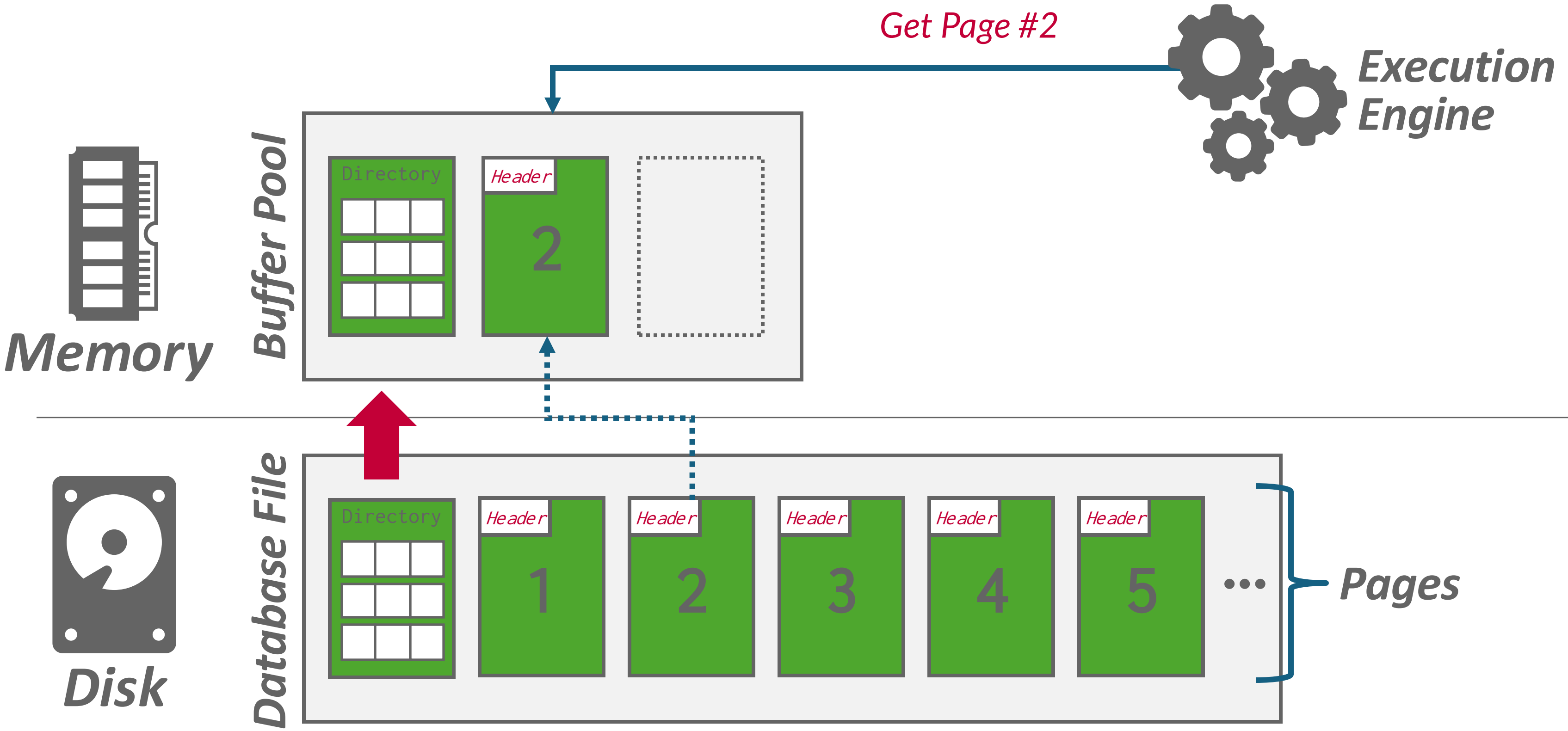
Disk-oriented DBMS



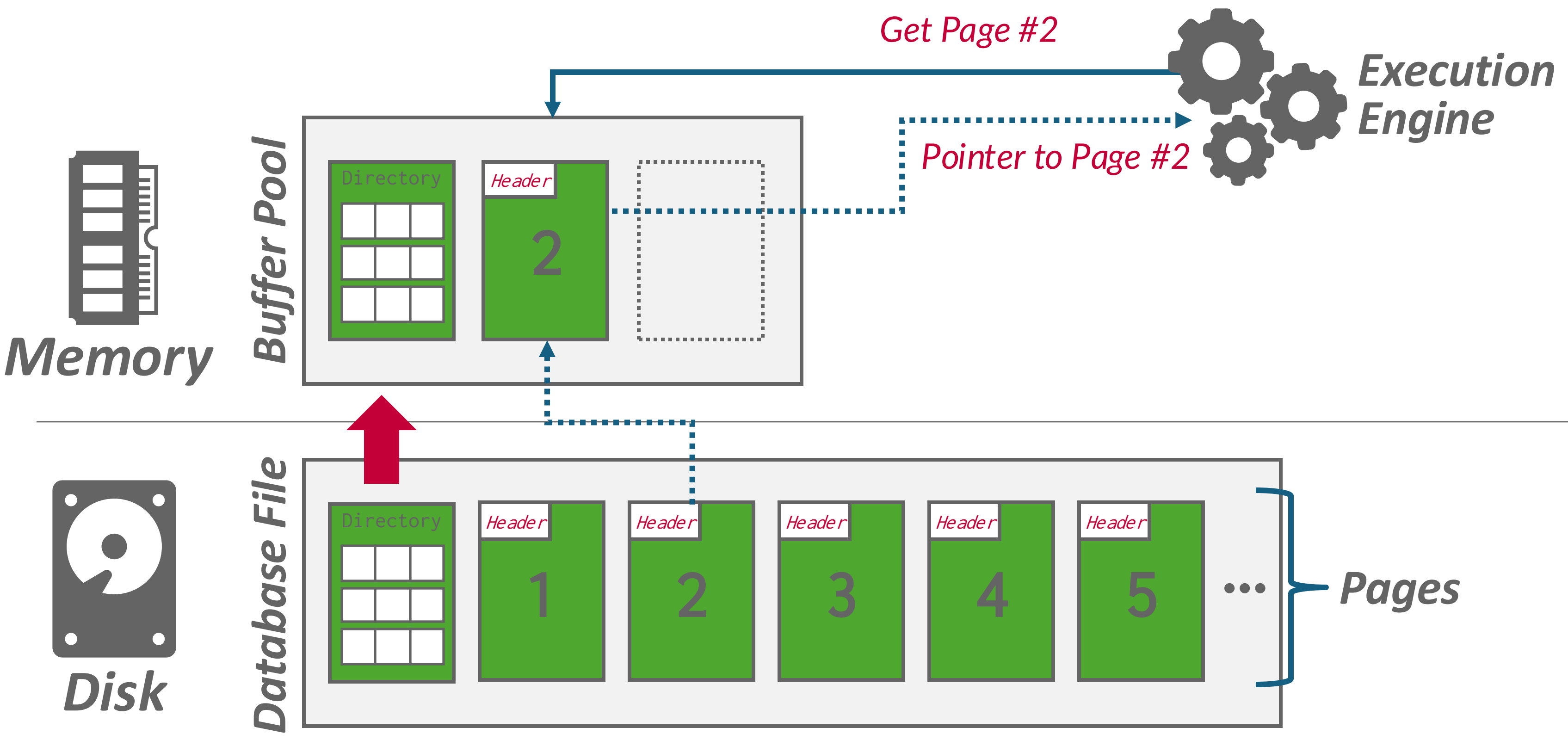
Disk-oriented DBMS



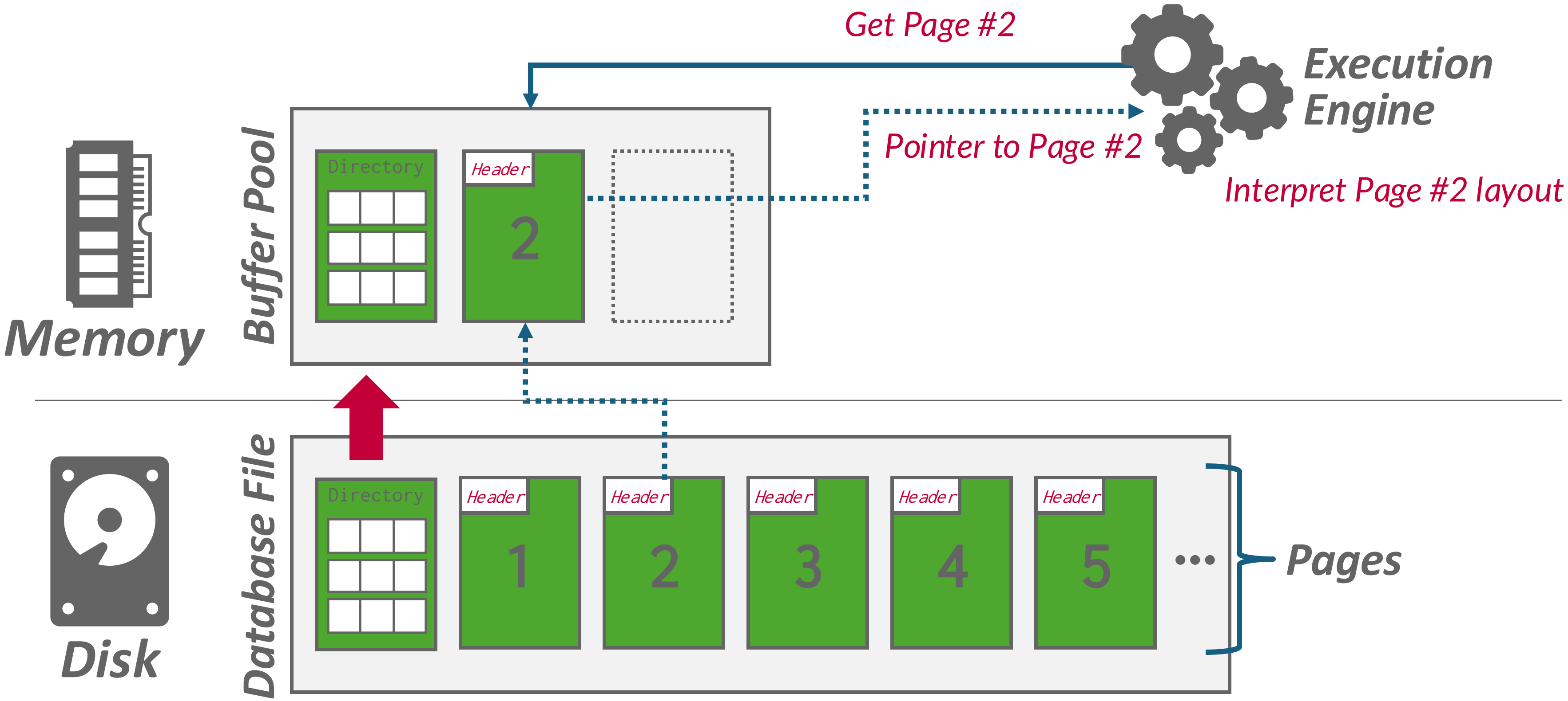
Disk-oriented DBMS



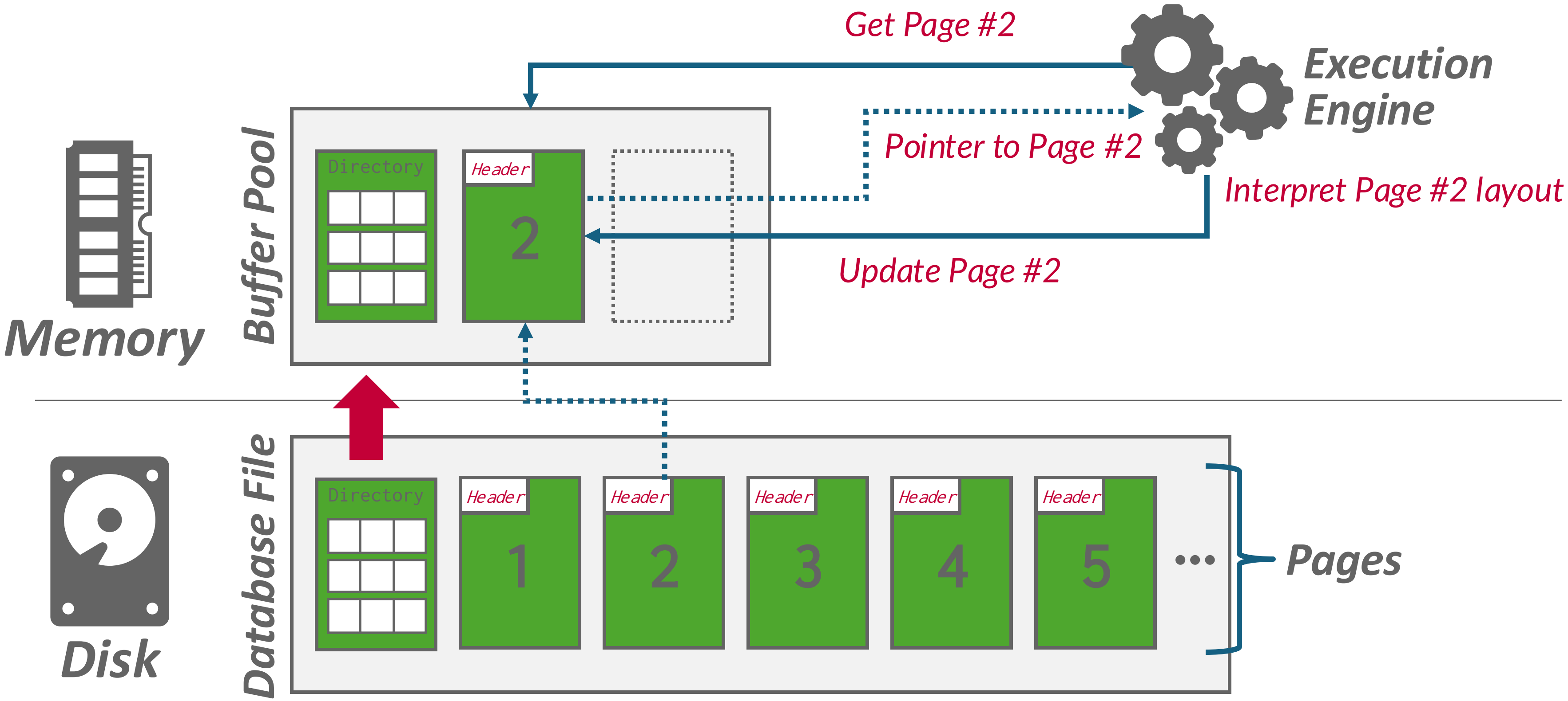
Disk-oriented DBMS



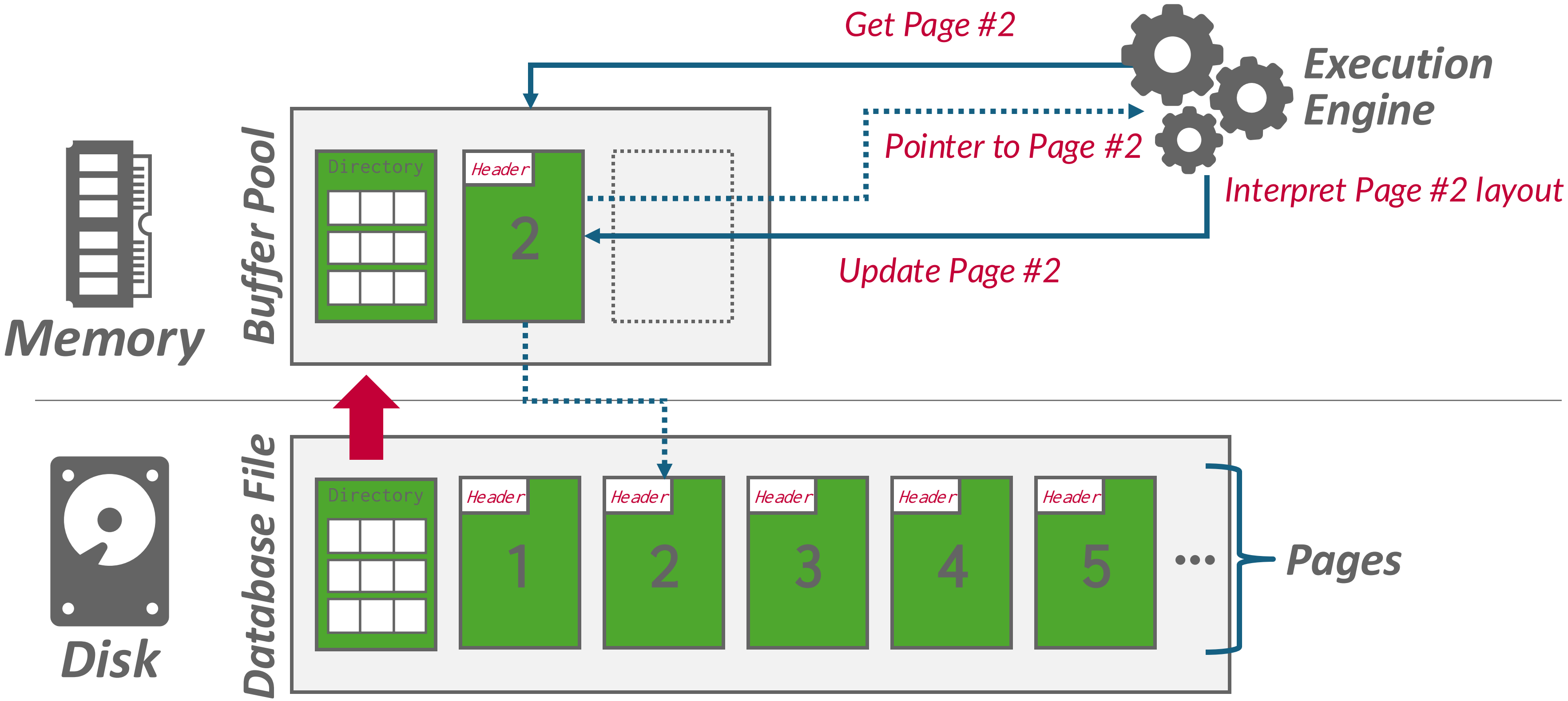
Disk-oriented DBMS



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.

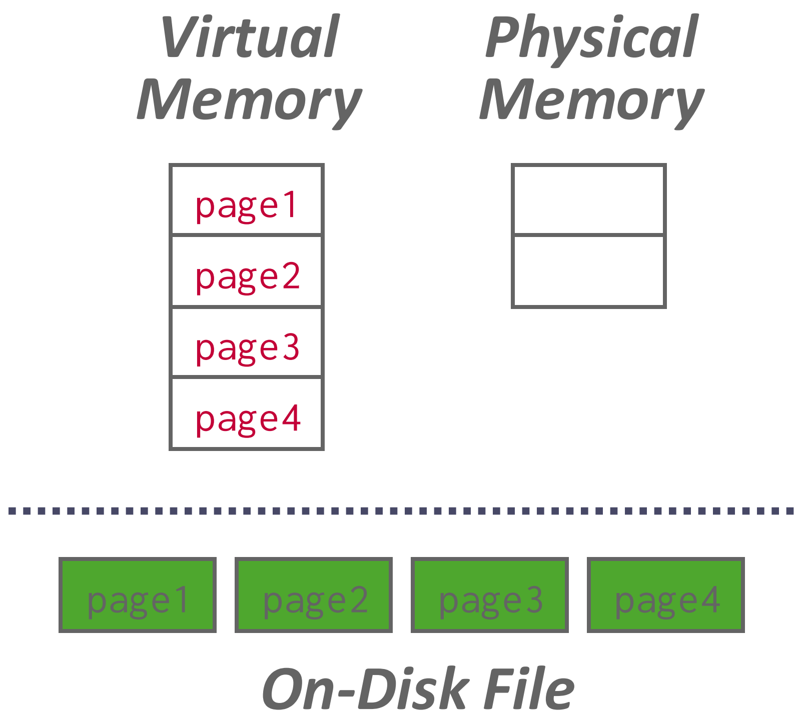
- OS is responsible for moving file pages in and out of memory, so the DBMS doesn't need to worry about it.

On-Disk File

Why Not Use the OS?

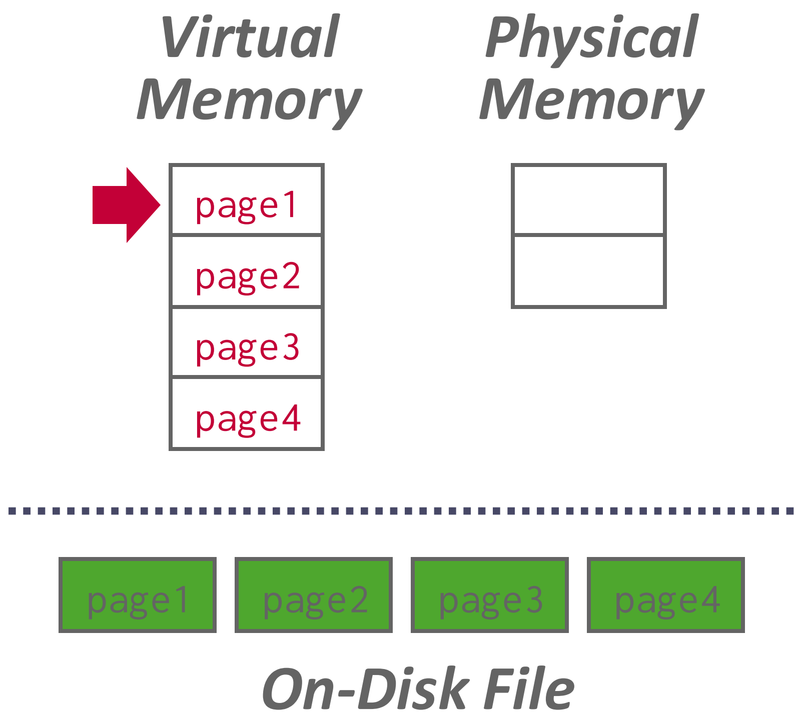
- The DBMS can use memory mapping (mmap) to store the contents of a file into the address space of a program.

- 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?

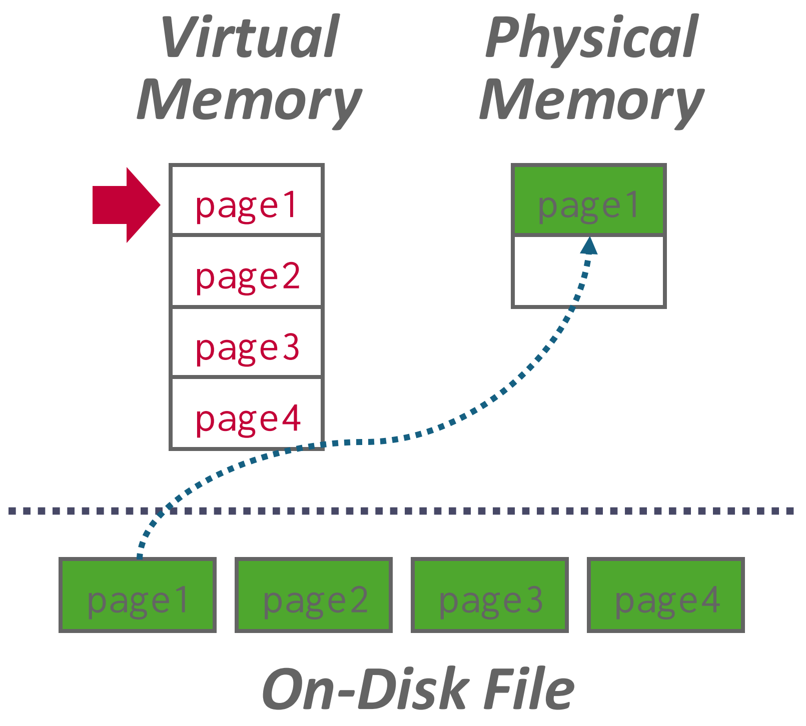
- The DBMS can use memory mapping (mmap) to store the contents of a file into the address space of a program.
- OS is responsible for moving file pages in and out of memory, so the DBMS doesn't need to worry about it.



Why Not Use the OS?

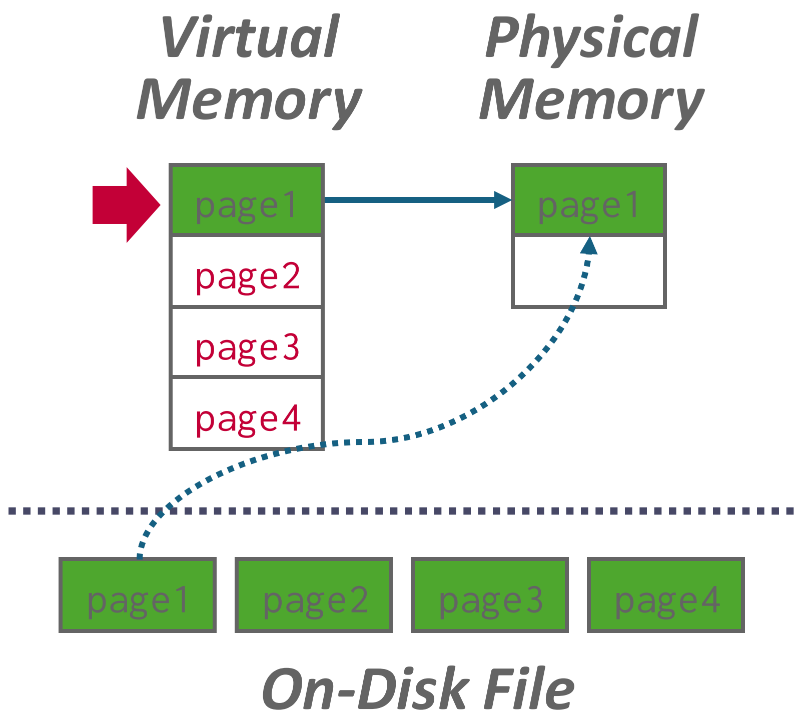
- The DBMS can use memory mapping (mmap) to store the contents of a file into the address space of a program.

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



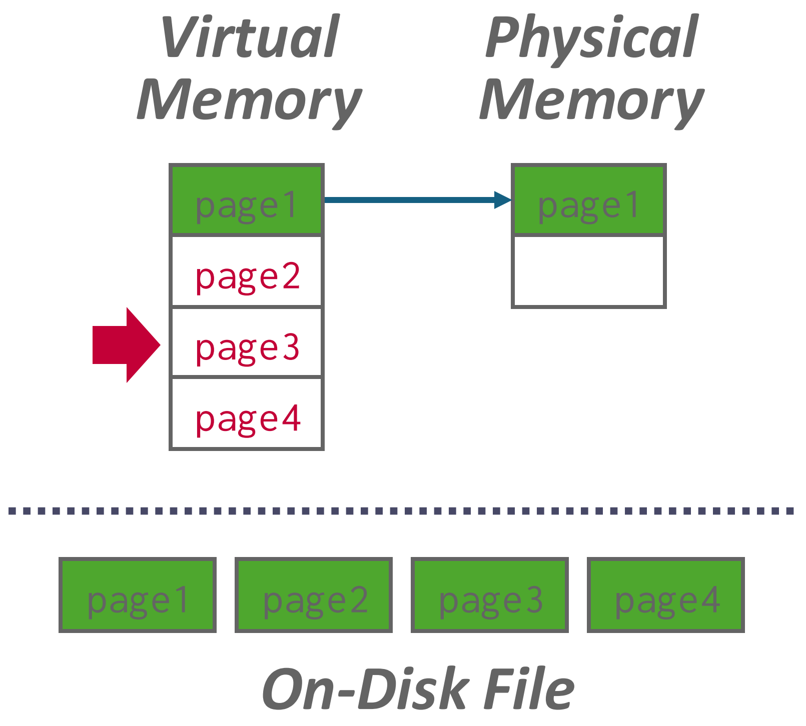
Why Not Use the OS?

- 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?

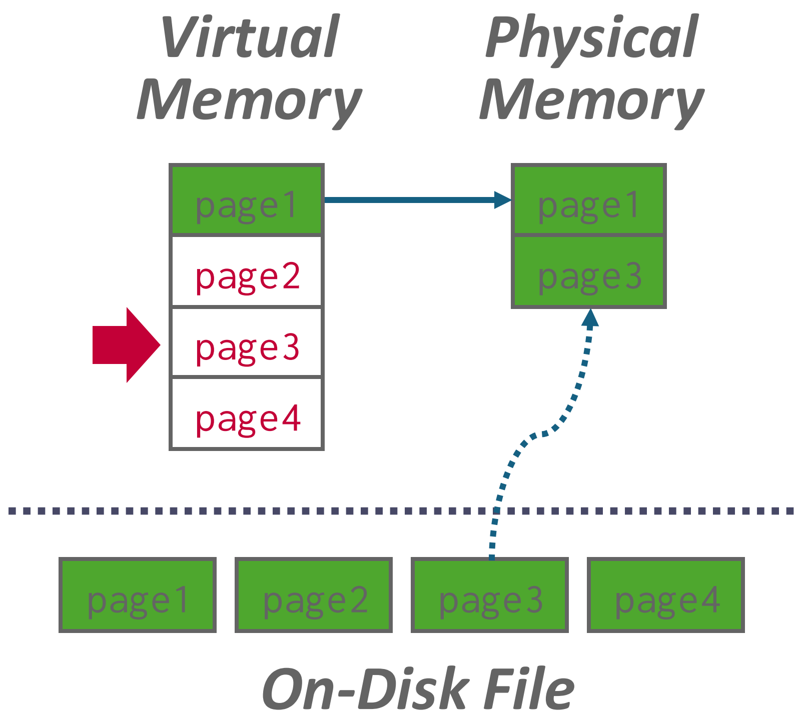
- The DBMS can use memory mapping (mmap) to store the contents of a file into the address space of a program.
- OS is responsible for moving file pages in and out of memory, so the DBMS doesn't need to worry about it.



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.

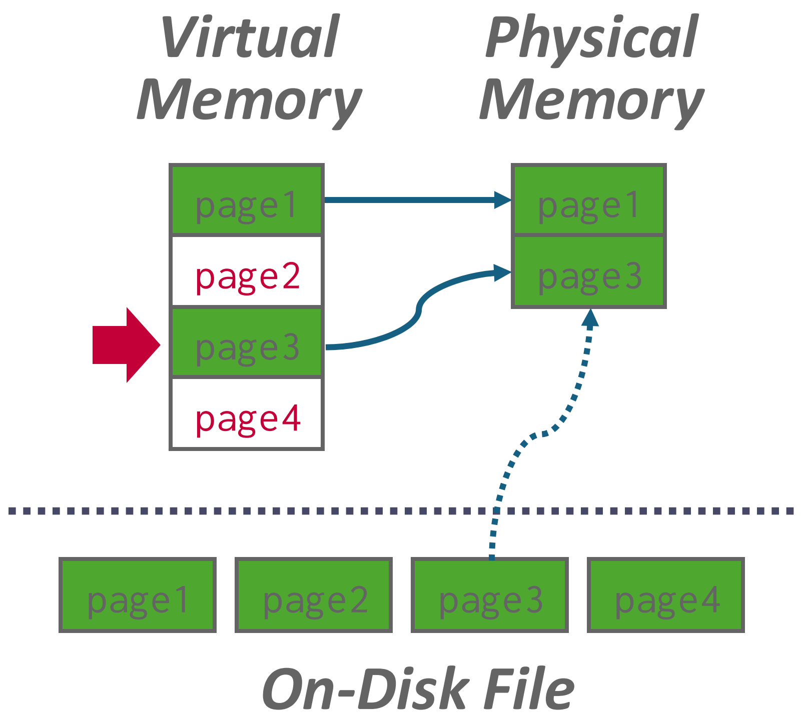
- OS is responsible for moving file pages in and out of memory, so the DBMS doesn't need to worry about it.



Why Not Use the OS?

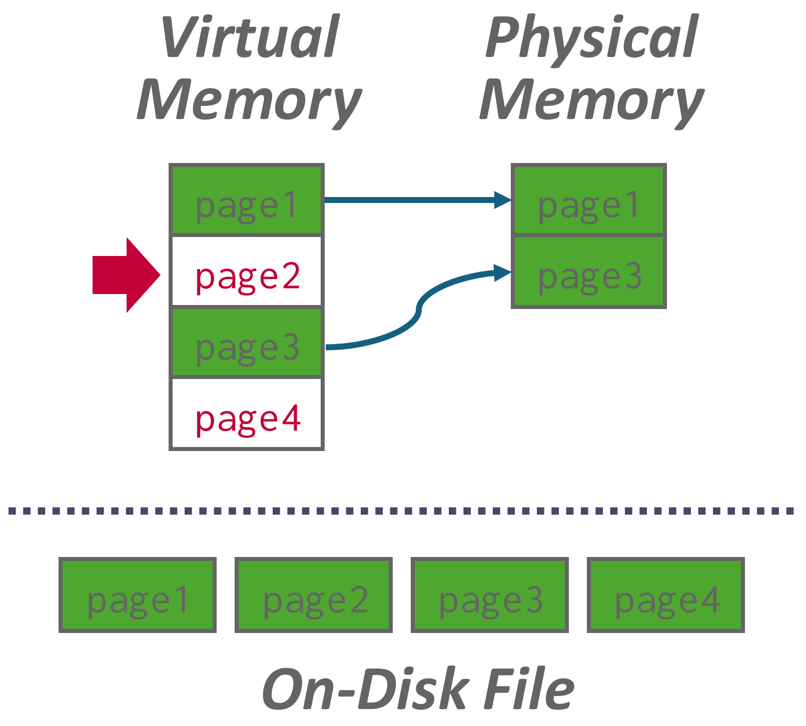
- The DBMS can use memory mapping (mmap) to store the contents of a file into the address space of a program.

- 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?

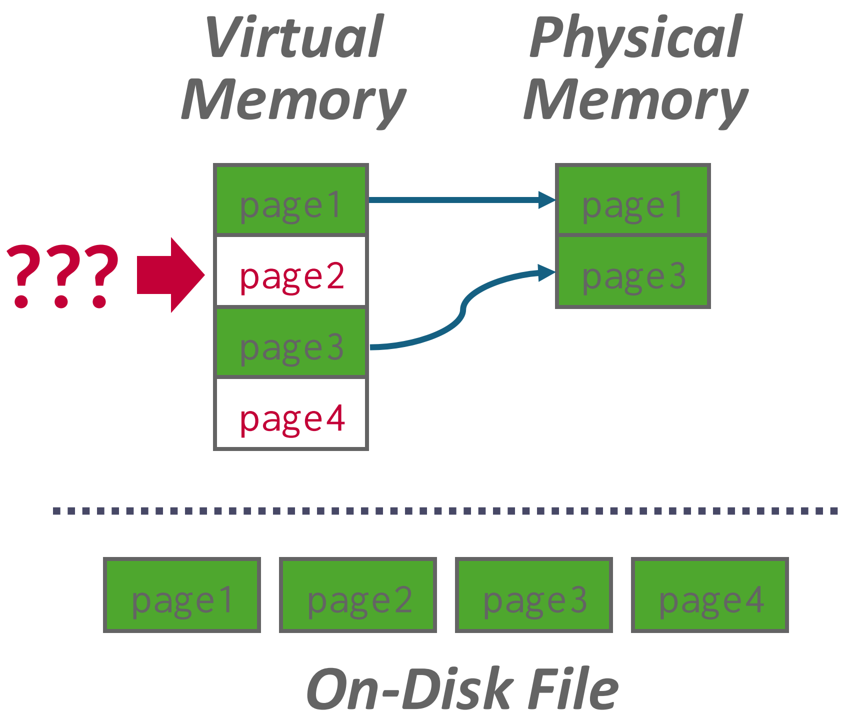
- The DBMS can use memory mapping (mmap) to store the contents of a file into the address space of a program.
- OS is responsible for moving file pages in and out of memory, so the DBMS doesn't need to worry about it.



Why Not Use the OS?

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

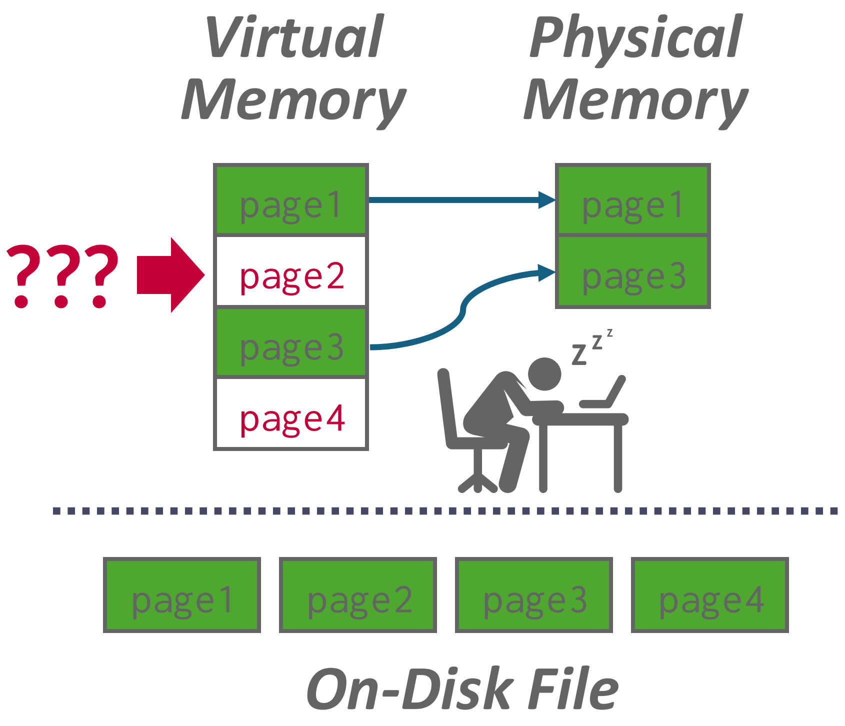
- OS is responsible for moving file pages in and out of memory, so the DBMS doesn't need to worry about it.



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 reasonably well for read-only access.
  It is complicated when there are multiple writers…

SCHOOL OF DATA SCIENCE

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.

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.

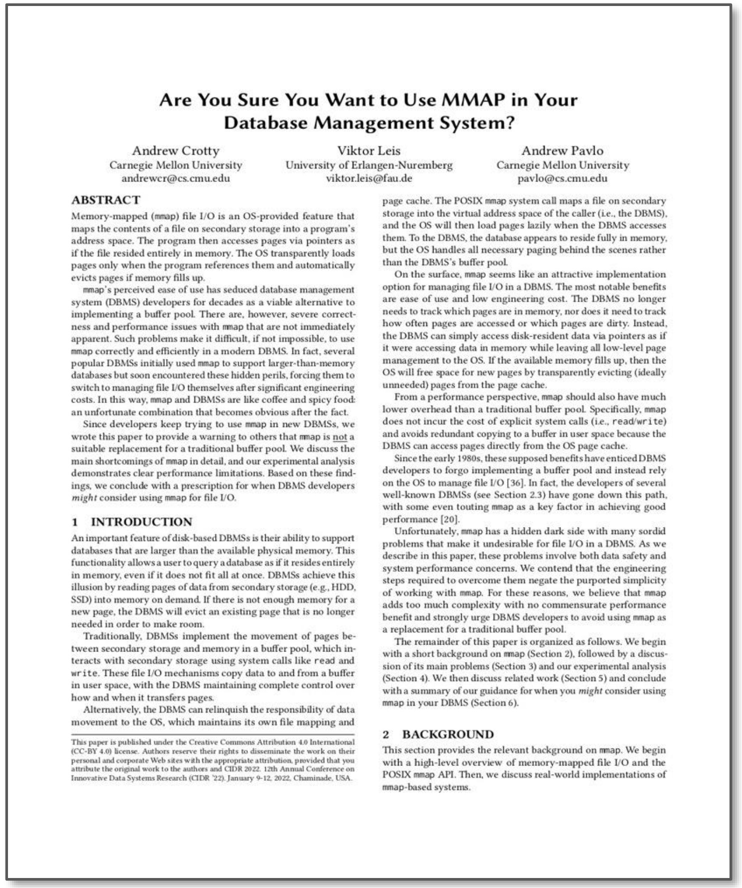
Full Usage

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 **<u>not</u>** 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 <u>not</u> 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.

Database Storage

- **Problem #1:** How the DBMS represents the database in files on disk.
- **Problem #2:** How the DBMS manages its memory and moves data back-and-forth from disk.

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.

This Lecture

- File Storage
- Page Layout
- Tuple Layout

File Storage

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.

- 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.

- Assume that if there is replication (for fault tolerance), it happens outside the core storage manager function.

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.

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

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.

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.

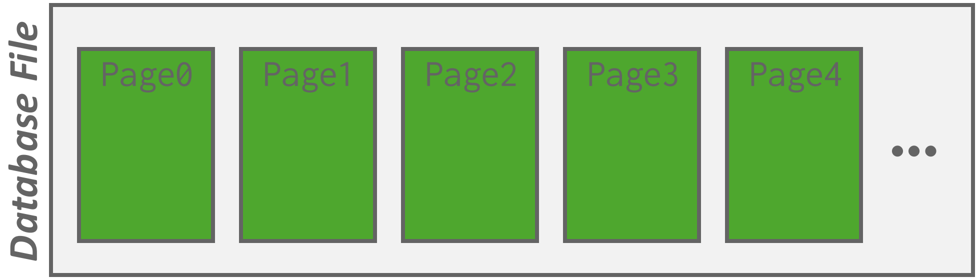
Heap File

- A heap file is an unordered collection of pages with tuples 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 track what pages exist in multiple files and which ones have free space.

SCHOOL OF DATA SCIENCE

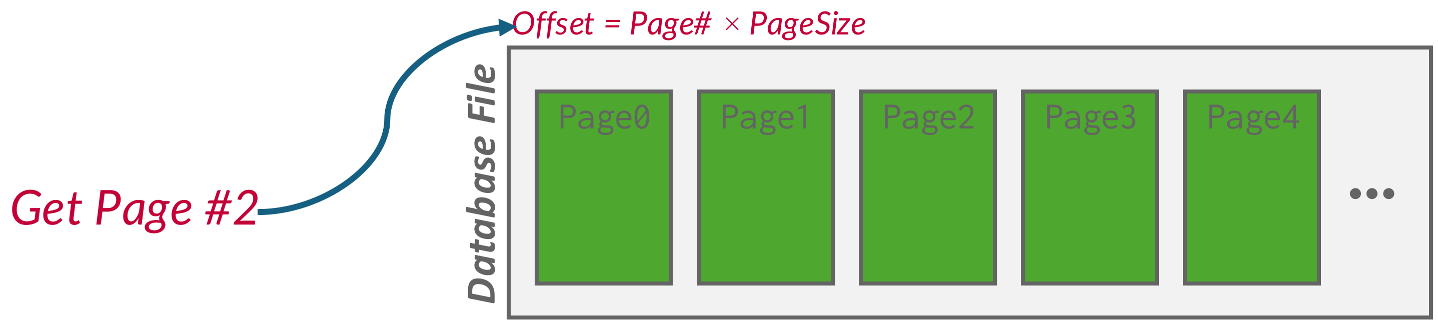
Heap File

- A heap file is an unordered collection of pages with tuples 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 track what pages exist in multiple files and which ones have free space.



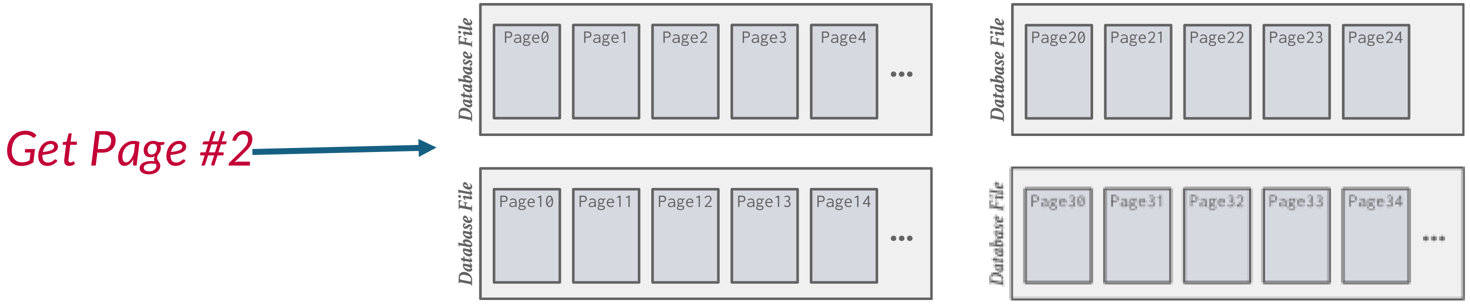
Heap File

- A <u>heap file</u> is an unordered collection of pages with tuples 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 track what pages exist in multiple files and which ones have free space.



Heap File

- A <u>heap file</u> is an unordered collection of pages with tuples 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 track what pages exist in multiple files and which ones have free space.



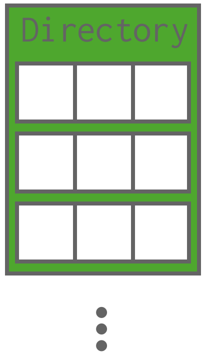
Heap File

- A heap file is an unordered collection of pages with tuples 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 track what pages exist in multiple files and which ones have free space.

SCHOOL OF DATA SCIENCE

Heap File: Page Directory

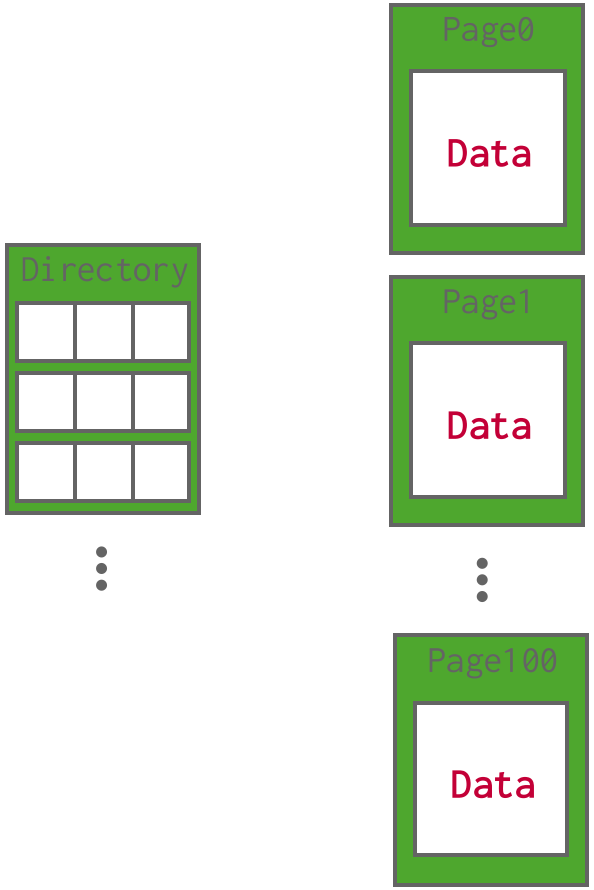
- The DBMS maintains special pages that tracks the location of data pages in the database files.
  - Must make sure that the directory pages are in sync with the data pages.



- The directory also records meta-data about available space:
  - The number of free slots per page.
  - List of free / empty pages.

Heap File: Page Directory

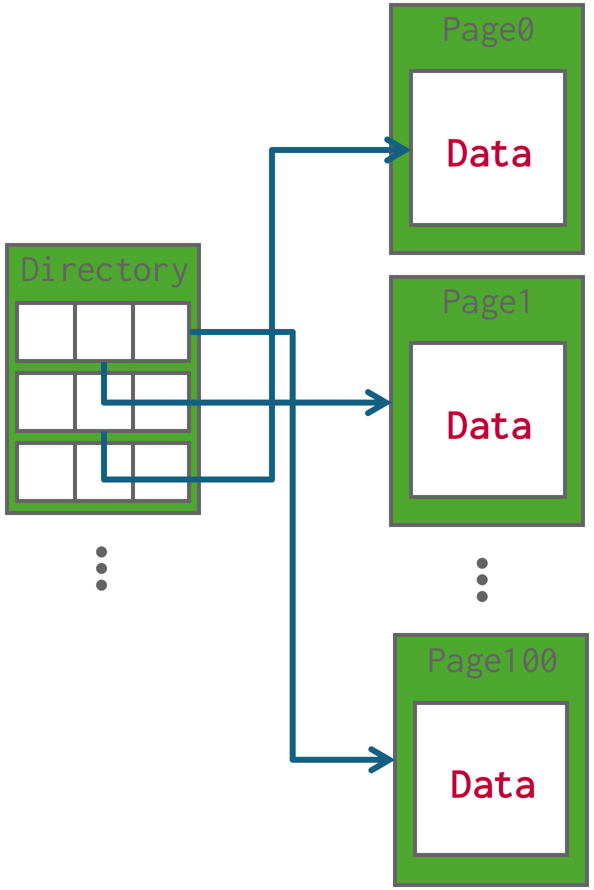
- The DBMS maintains special pages that tracks the location of data pages in the database files.
  - Must make sure that the directory pages are in sync with the data pages.
- The directory also records meta-data about available space:
  - The number of free slots per page.
  - List of free / empty pages.



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.



Page Layout

File Storage

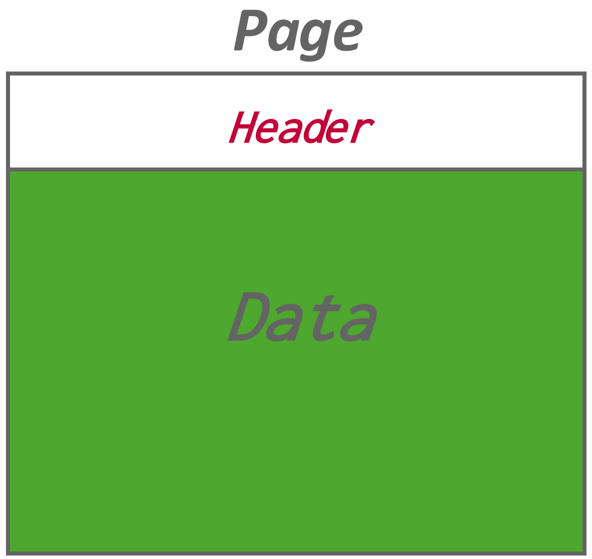
Page Layout

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).



SCHOOL OF DATA SCIENCE

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**.

Approach #1: Tuple-oriented Storage

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.**

$\boxed{\textbf{Approach \#1: Tuple-oriented Storage}}$ ← Today

Approach #2: Log-structured Storage

Approach #3: Index-organized Storage

SCHOOL OF DATA SCIENCE

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.**

Approach #1: Tuple-oriented Storage

Approach #2: Log-structured Storage

Approach #3: Index-organized Storage

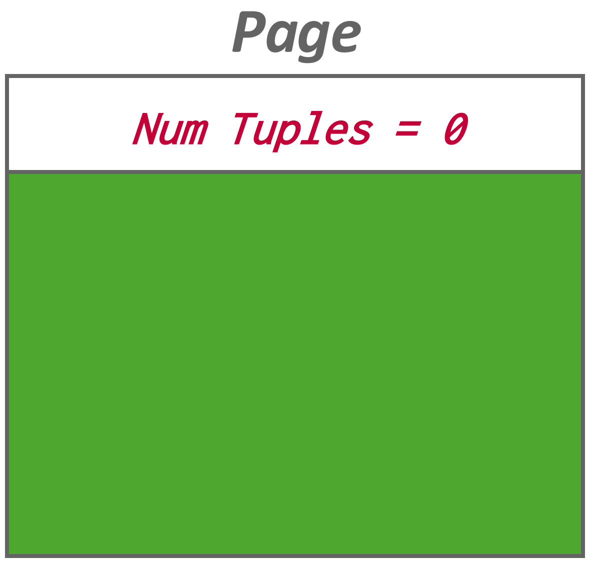
Omitted

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?



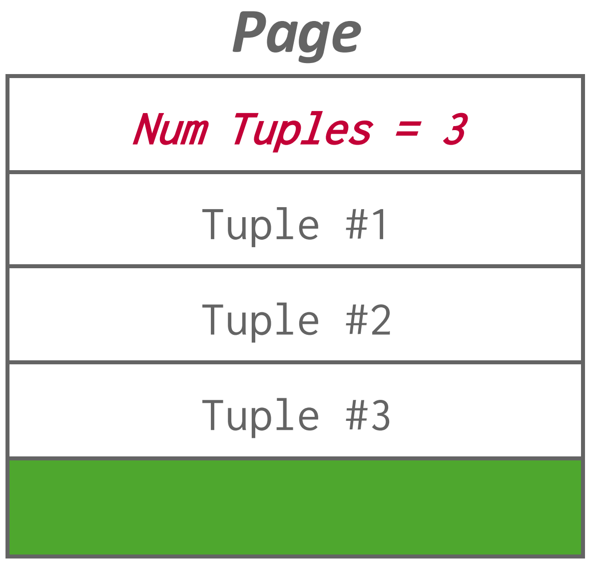
Tuple-oriented Storage

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?

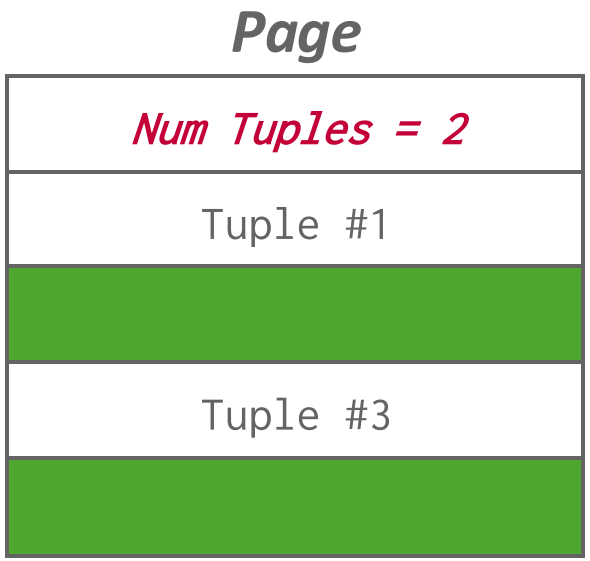


Tuple-oriented Storage

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?

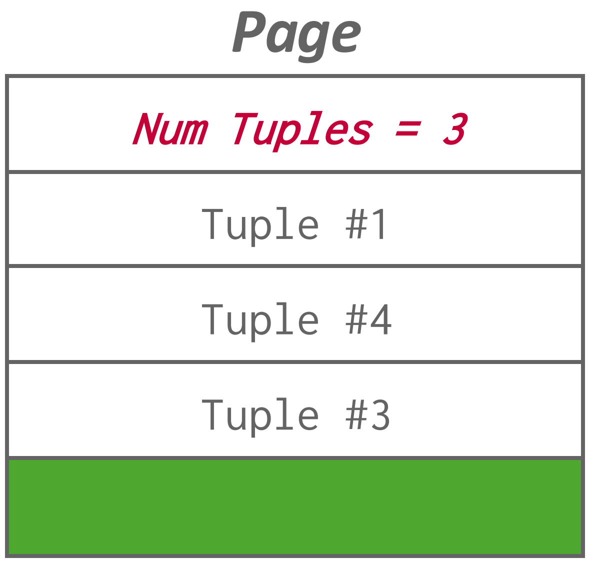


Tuple-oriented Storage

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?

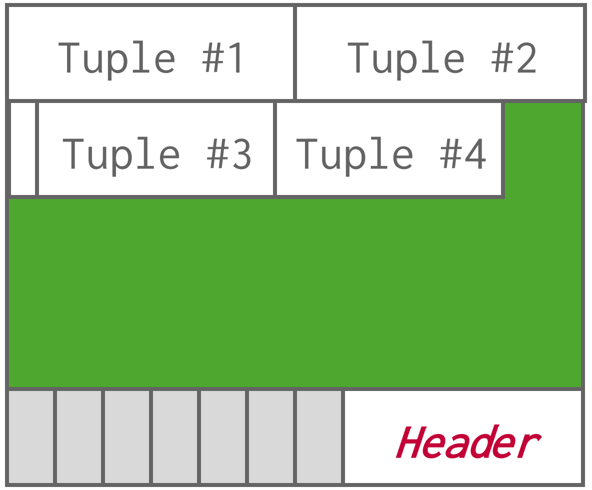


Slotted Pages

- 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

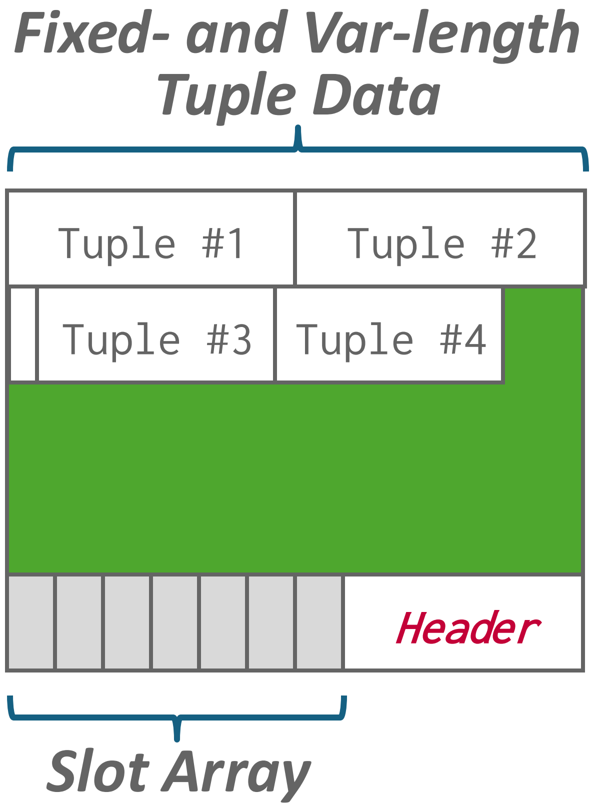
- The most common layout scheme is called <u>slotted pages</u>.

- 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

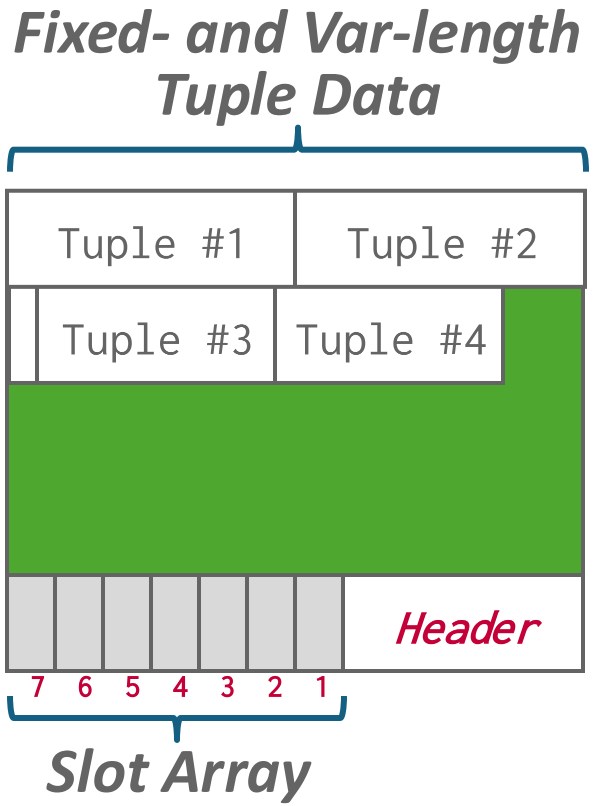
- The most common layout scheme is called <u>slotted pages</u>.

- 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.



SCHOOL OF DATA SCIENCE

Slotted Pages

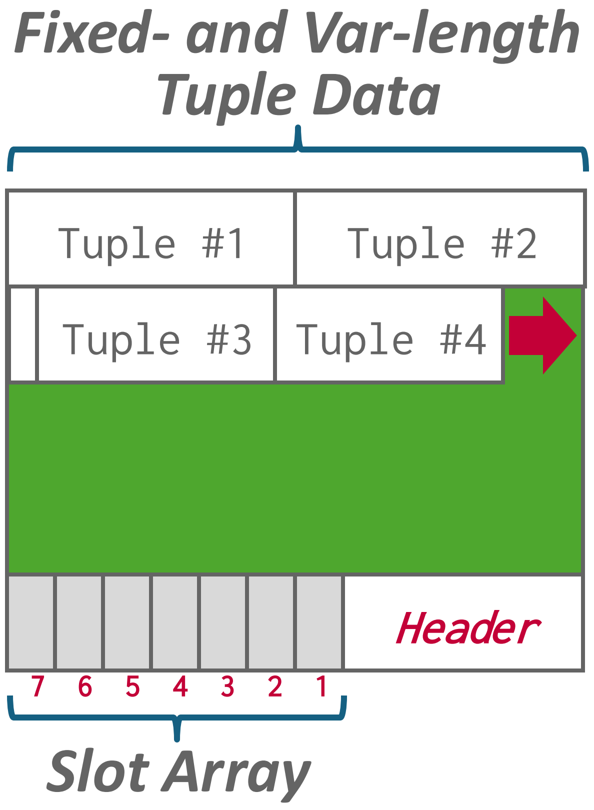
- The most common layout scheme is called <u>slotted pages</u>.

- 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.

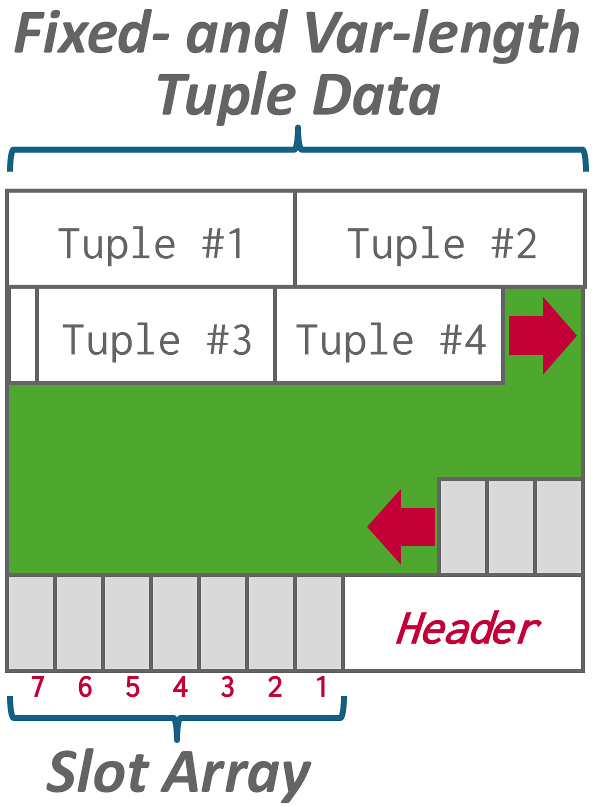


Slotted Pages

- The most common layout scheme is called <u>slotted pages</u>.
- 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

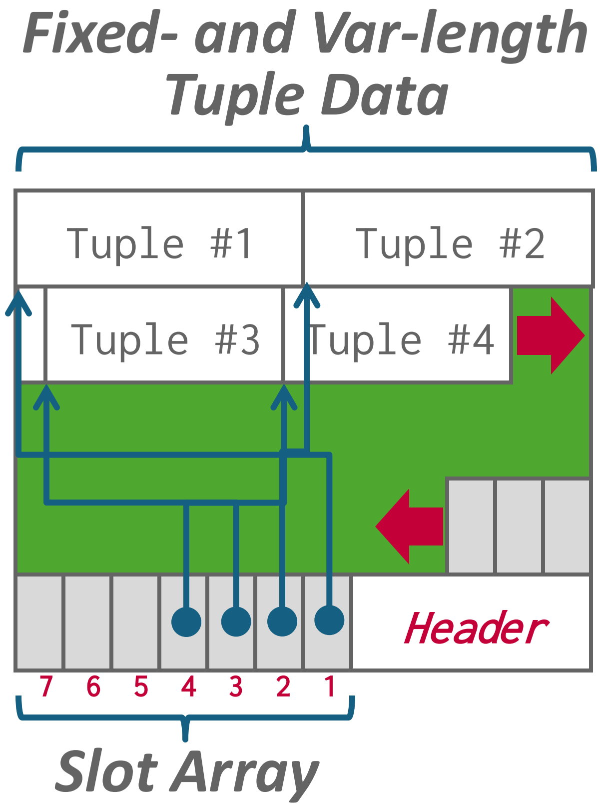
- The most common layout scheme is called <u>slotted pages</u>.

- 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

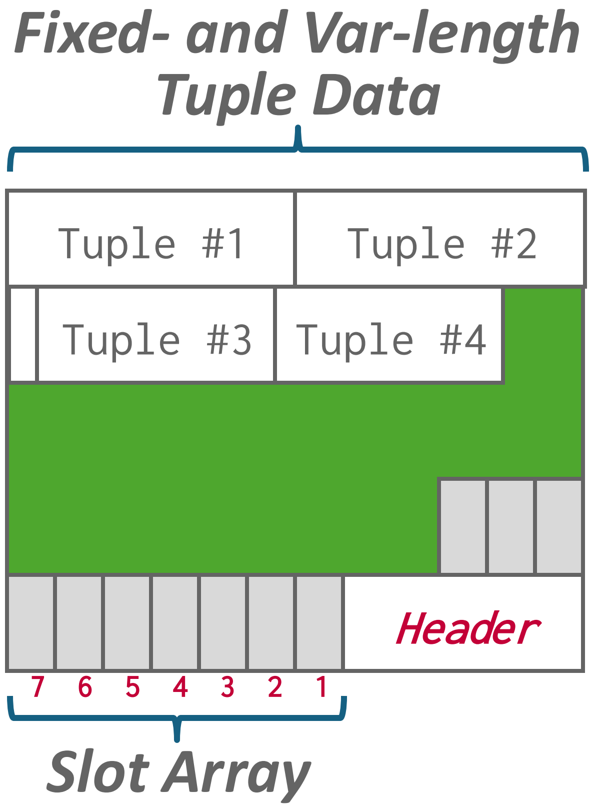
- The most common layout scheme is called <u>slotted pages</u>.

- 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.

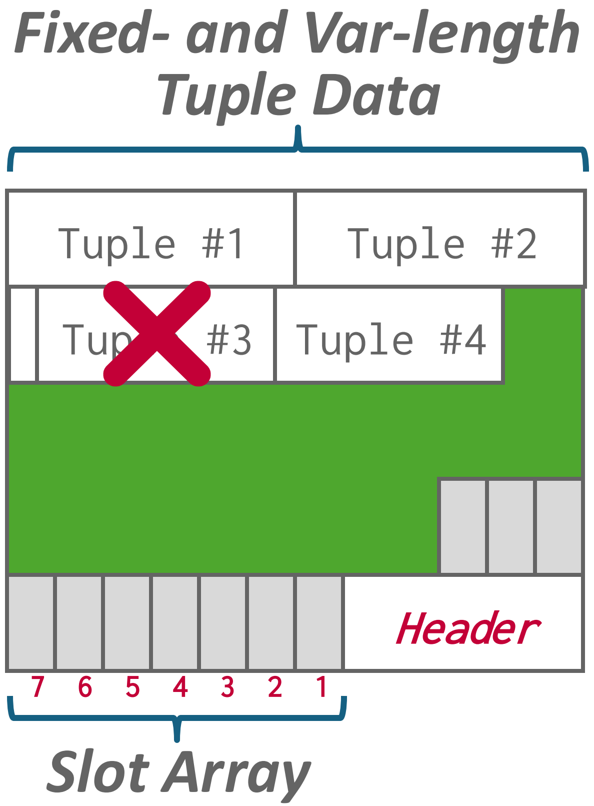


Slotted Pages

- 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.

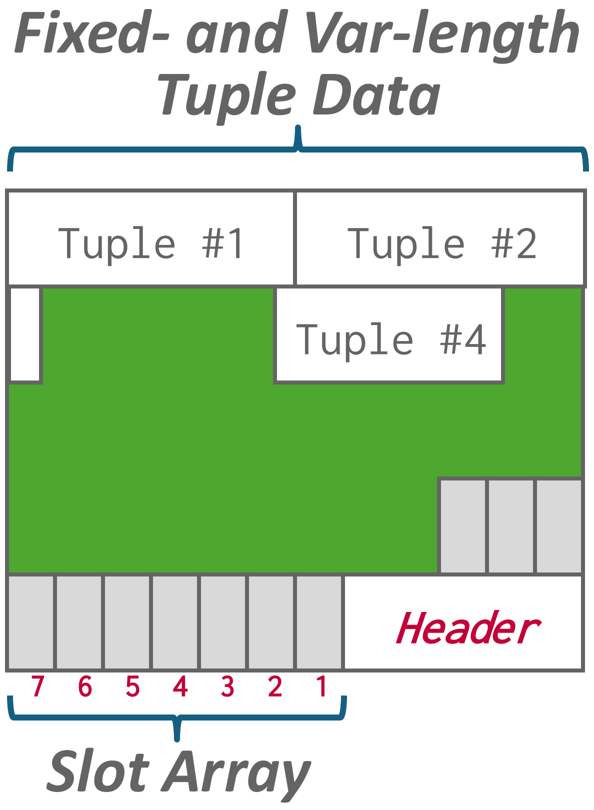


Slotted Pages

- The most common layout scheme is called <u>slotted pages</u>.
- 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

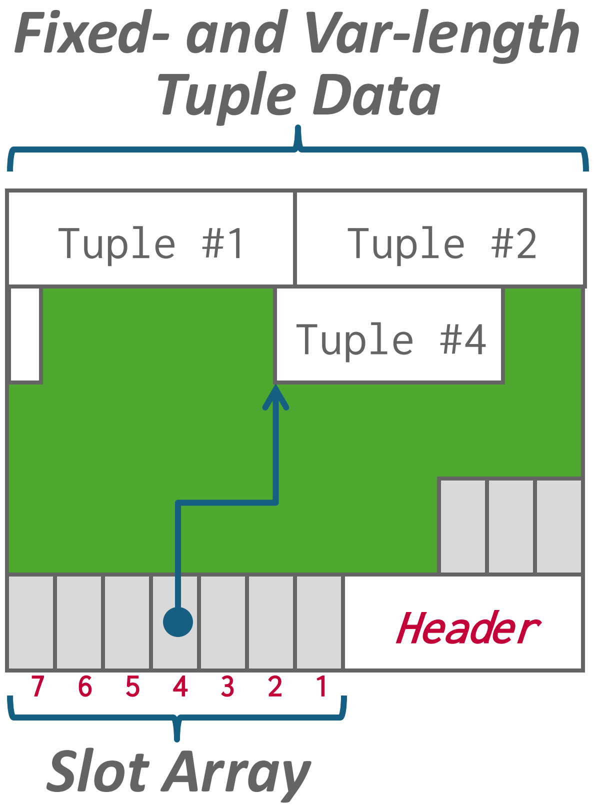
- The most common layout scheme is called <u>slotted pages</u>.

- 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.

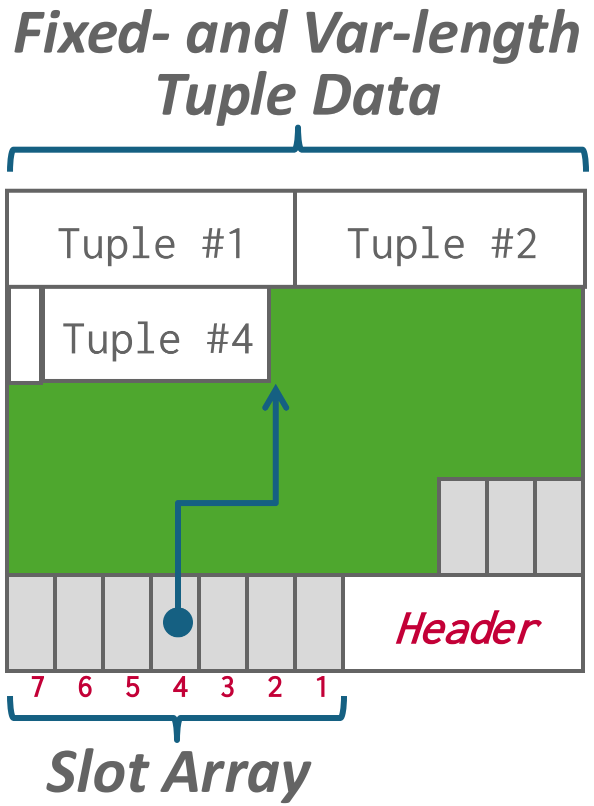


Slotted Pages

- 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.



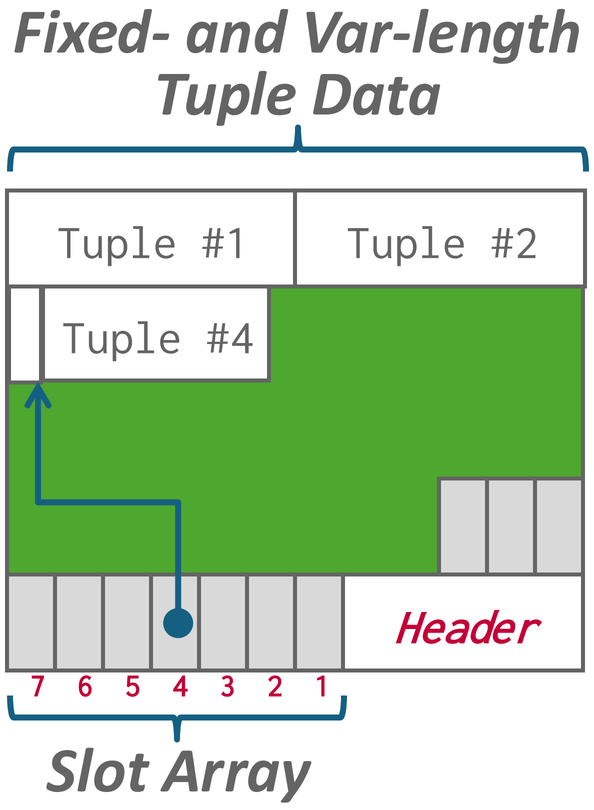
SCHOOL OF
DATA SCIENCE

Slotted Pages

- The most common layout scheme is called <u>slotted pages</u>.
- 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 representing its physical location in the database.

→ File Id, Page Id, Slot #

→ Most DBMSs do not store IDs in tuples.

→ SQLite uses ROWID as the true primary key and stores it as a hidden attribute.

- Applications should never rely on these IDs to mean anything.

Record IDs

- The DBMS assigns each logical tuple a unique <u>record identifier</u> representing its physical location in the database.

→ File Id, Page Id, Slot #

→ Most DBMSs do not store IDs in tuples.

→ SQLite uses <u>ROWID</u> as the true primary key and stores it as a hidden attribute.

- Applications should <u>never</u> 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)

Tuple Layout

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 <u>header</u> that contains meta-data about it.

→ Visibility info (concurrency control)

→ Bit Map for `NULL` values.

- We do <u>not</u> need to store meta-data about the schema.

Tuple Data

- Attributes are typically stored in the order specified in the DDL used to create the table.
- This is done for software engineering reasons (i.e., simplicity).
- However, it might be more efficient to lay them out differently.

Tuple

Header	a	b	c	d	e

```
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.

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

```
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.

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

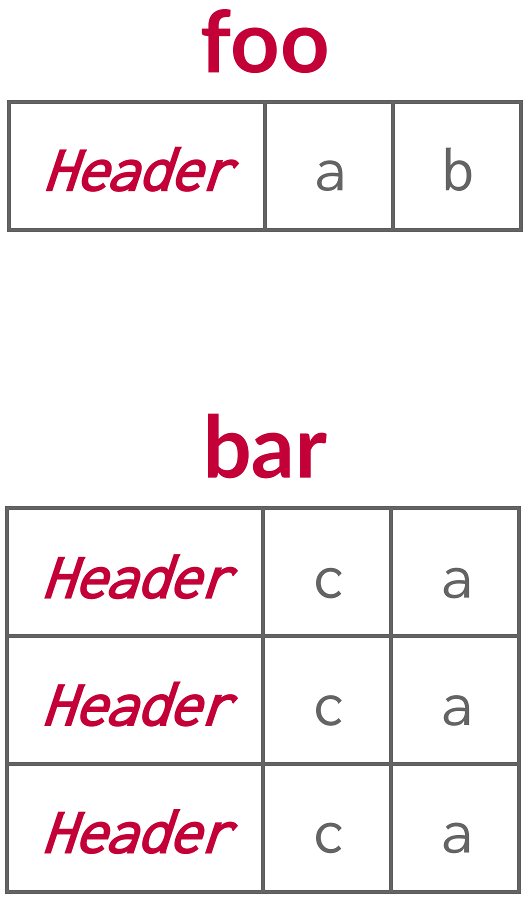
```
CREATE TABLE bar (
  c INT PRIMARY KEY,
  a INT
  ⮱REFERENCES foo (a),
);
```

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.



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;
```

foo

Header	a	b

bar

Header	c	a
Header	c	a
Header	c	a

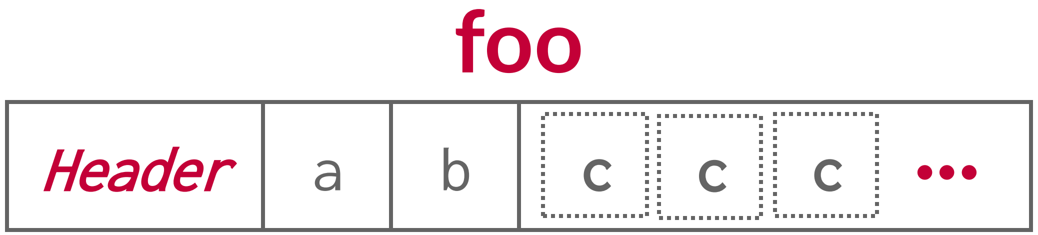
Denormalized Tuple Data

- 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;
```



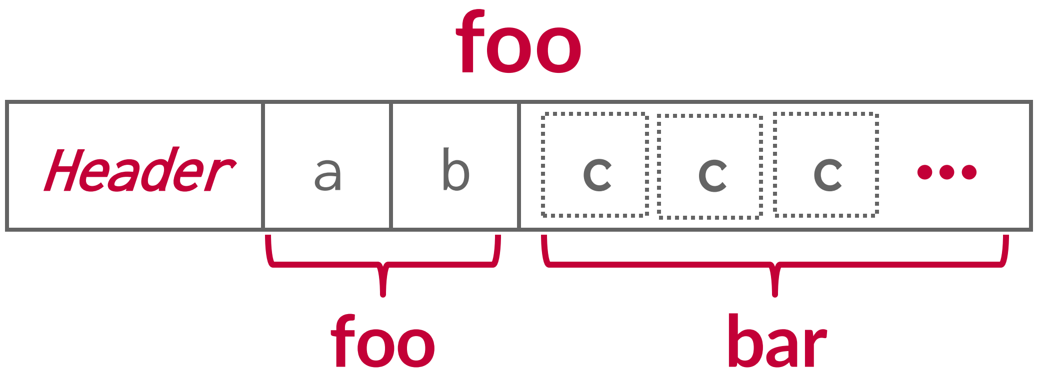
Denormalized Tuple Data

- 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.

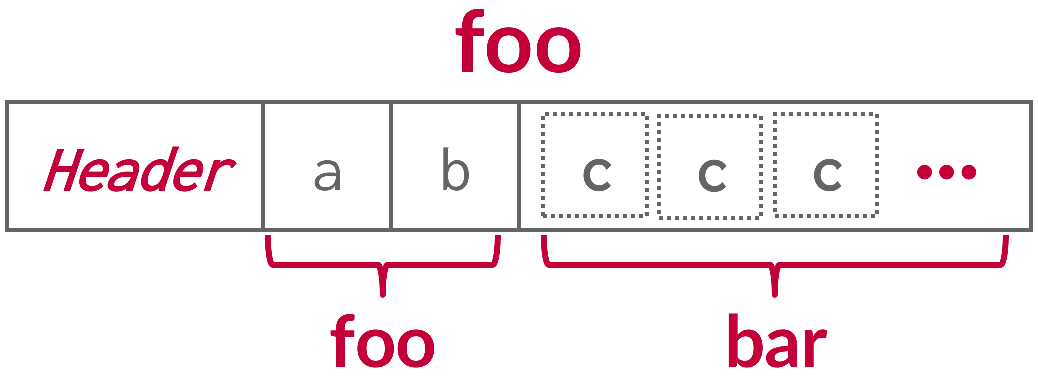
```
SELECT * FROM foo JOIN bar
    ON foo.a = bar.a;
```



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.



Denormalized Tuple Data

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

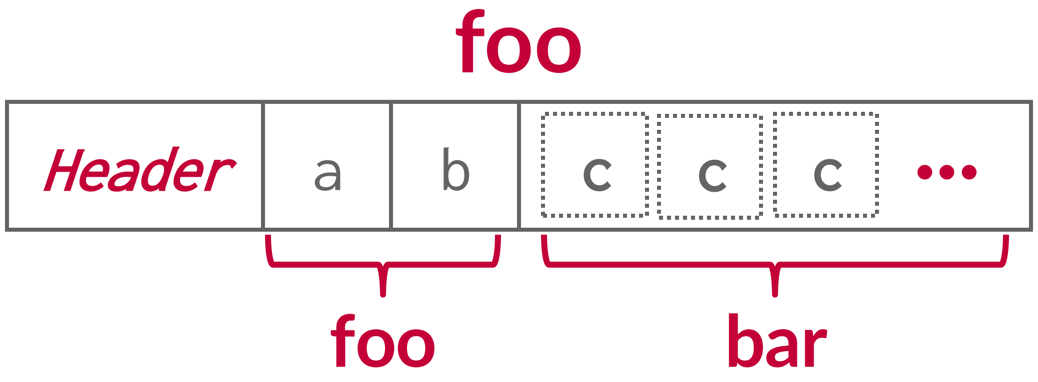
→ Potentially reduces the amount of I/O for common workload patterns.

→ 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.



Tuple Storage

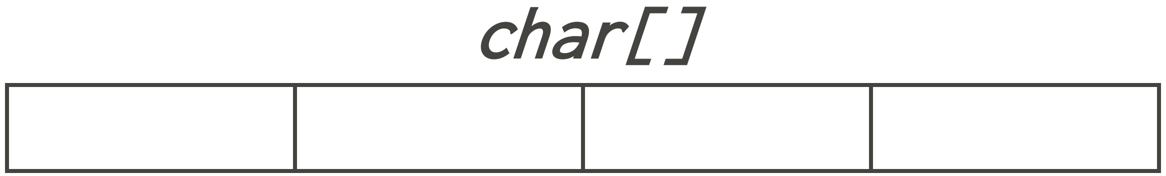
- A tuple is essentially a sequence of bytes.
- It is the job of the DBMS to interpret those bytes into attribute types and values.

- The DBMS's catalogs contain the schema information about tables that the system uses to figure out the tuple's layout.

Word-aligned Tuples

- All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  cdate TIMESTAMP,
  color CHAR(2),
  zipcode INT
);
```

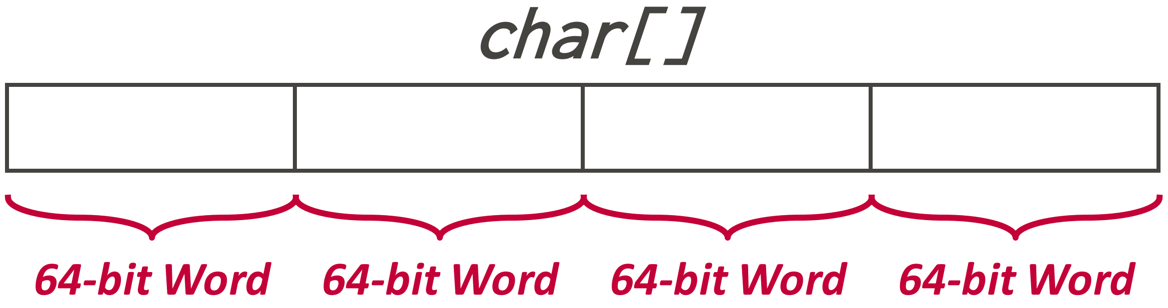


In the old days, the DBMS programmer had to worry about “unaligned word memory reference.”
Today: Processors handle it. It will read multiple words from memory, so it may have a performance impact.

Word-aligned Tuples

- All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  cdate TIMESTAMP,
  color CHAR(2),
  zipcode INT
);
```



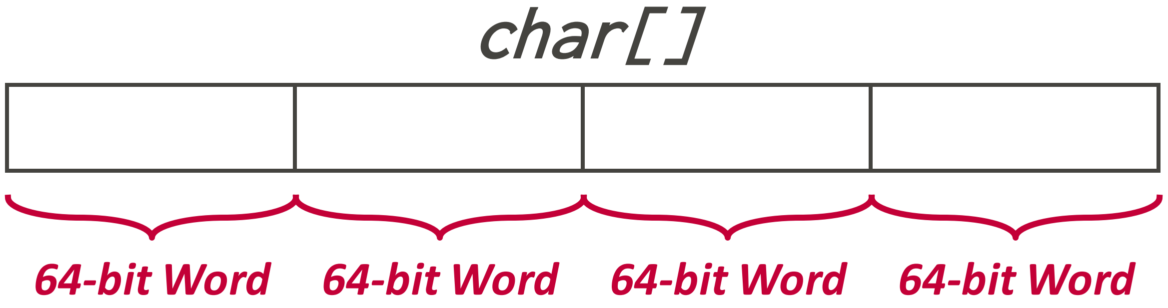
In the old days, the DBMS programmer had to worry about “unaligned word memory reference.”
Today: Processors handle it. It will read multiple words from memory, so it may have a performance impact.

Word-aligned Tuples

- All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

32-bits

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  cdate TIMESTAMP,
  color CHAR(2),
  zipcode INT
);
```



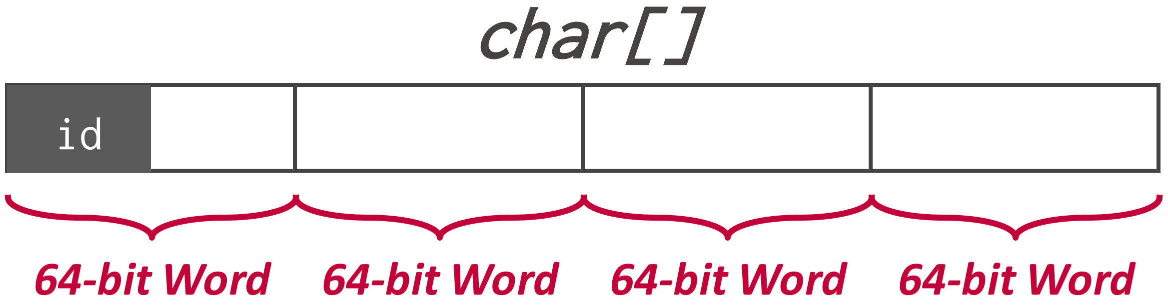
In the old days, the DBMS programmer had to worry about "unaligned word memory reference."
Today: Processors handle it. It will read multiple words from memory, so it may have a performance impact.

Word-aligned Tuples

- All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

32-bits

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  cdate TIMESTAMP,
  color CHAR(2),
  zipcode INT
);
```

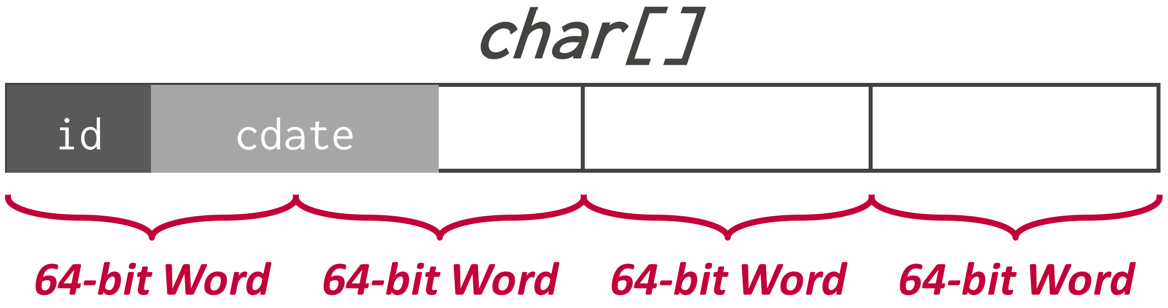


In the old days, the DBMS programmer had to worry about “unaligned word memory reference.”
Today: Processors handle it. It will read multiple words from memory, so it may have a performance impact.

Word-aligned Tuples

- All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

```
        CREATE TABLE foo (
32-bits   id INT PRIMARY KEY,
64-bits   cdate TIMESTAMP,
          color CHAR(2),
          zipcode INT
        );
```

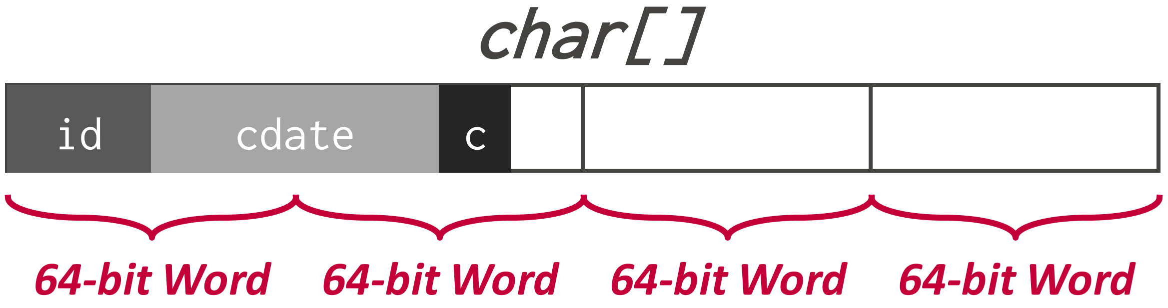


In the old days, the DBMS programmer had to worry about "unaligned word memory reference."
Today: Processors handle it. It will read multiple words from memory, so it may have a performance impact.

Word-aligned Tuples

- All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

```
        CREATE TABLE foo (
32-bits   id INT PRIMARY KEY,
64-bits   cdate TIMESTAMP,
16-bits   color CHAR(2),
          zipcode INT
        );
```

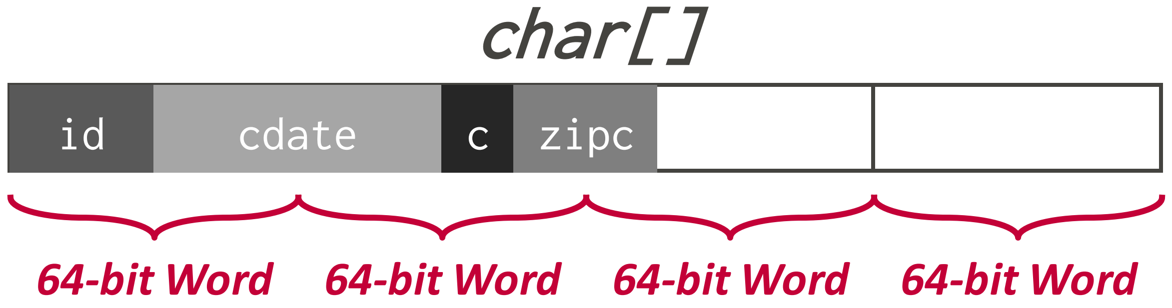


In the old days, the DBMS programmer had to worry about “unaligned word memory reference.”
Today: Processors handle it. It will read multiple words from memory, so it may have a performance impact.

Word-aligned Tuples

- All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

	CREATE TABLE foo (
32-bits	id INT PRIMARY KEY,
64-bits	cdate TIMESTAMP,
16-bits	color CHAR(2),
32-bits	zipcode INT
	);

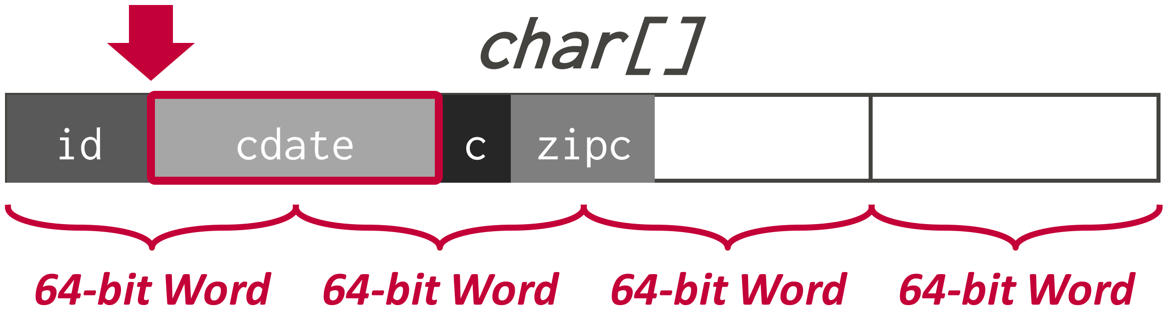


In the old days, the DBMS programmer had to worry about "unaligned word memory reference."
Today: Processors handle it. It will read multiple words from memory, so it may have a performance impact.

Word-aligned Tuples

- All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

```
        CREATE TABLE foo (
32-bits   id INT PRIMARY KEY,
64-bits   cdate TIMESTAMP,
16-bits   color CHAR(2),
32-bits   zipcode INT
        );
```

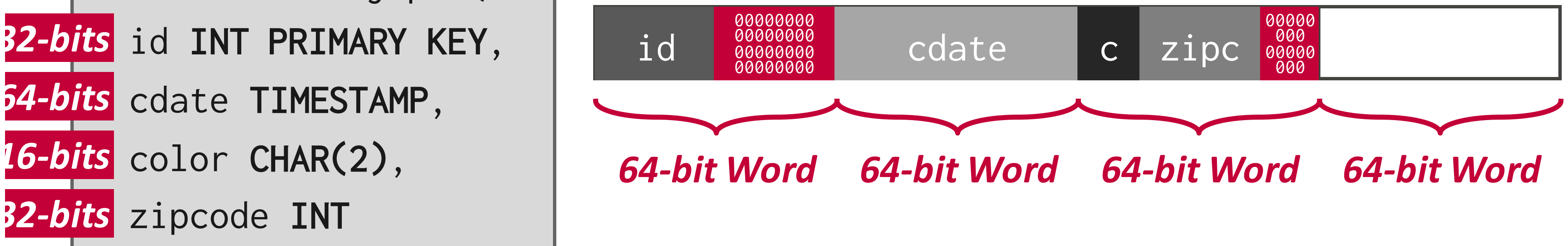


In the old days, the DBMS programmer had to worry about "unaligned word memory reference."
Today: Processors handle it. It will read multiple words from memory, so it may have a performance impact.

Word-alignment: Padding

- Add empty bits after attributes to ensure that tuple is word aligned.

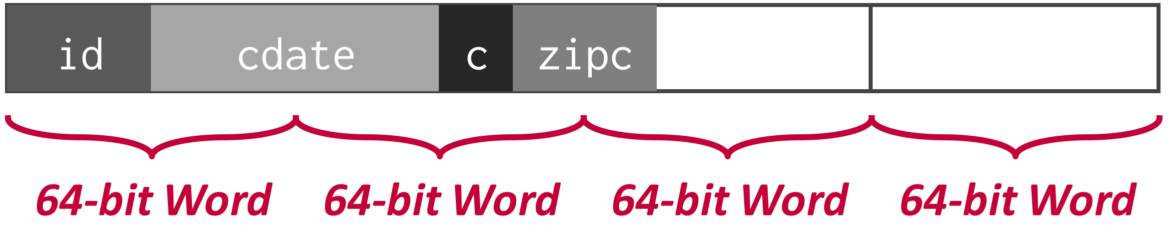
```
CREATE TABLE dj2pl (
  id INT PRIMARY KEY,
  cdate TIMESTAMP,
  color CHAR(2),
  zipcode INT
);
```



Word-alignment: Reordering

- Switch the order of attributes in the tuples' physical layout to make sure they are aligned.
  - May still have to use padding.

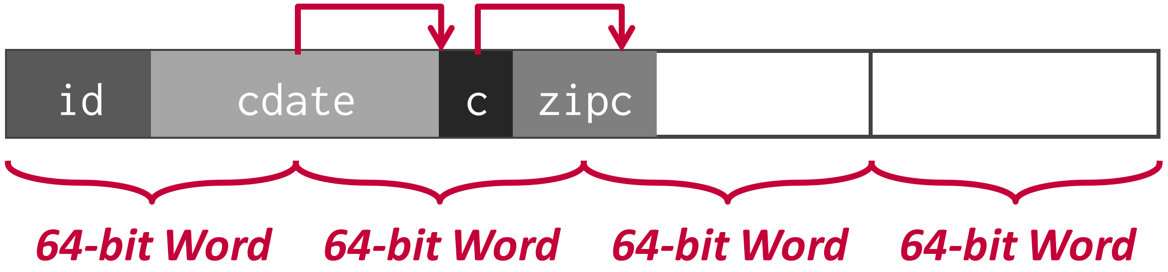
```
CREATE TABLE dj2pl (
32-bits  id INT PRIMARY KEY,
64-bits  cdate TIMESTAMP,
16-bits  color CHAR(2),
32-bits  zipcode INT
);
```



Word-alignment: Reordering

- Switch the order of attributes in the tuples' physical layout to make sure they are aligned.
  - May still have to use padding.

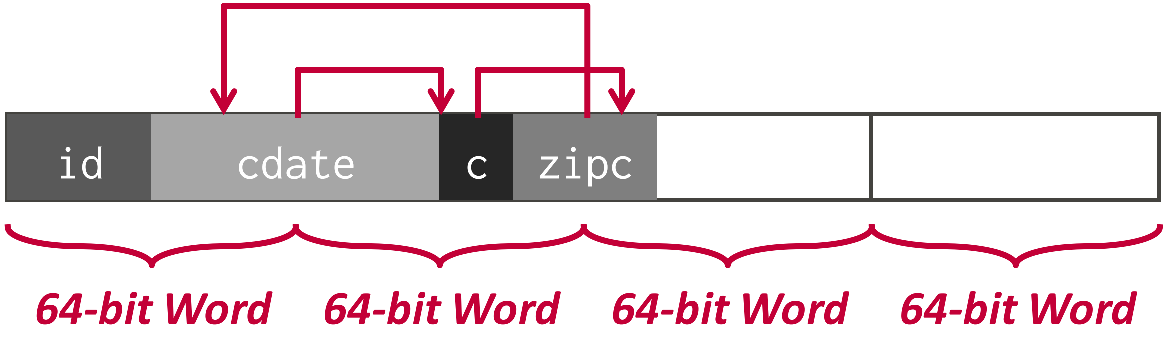
```
         CREATE TABLE dj2pl (
32-bits    id INT PRIMARY KEY,
64-bits    cdate TIMESTAMP,
16-bits    color CHAR(2),
32-bits    zipcode INT
         );
```



Word-alignment: Reordering

- Switch the order of attributes in the tuples' physical layout to make sure they are aligned.
  - May still have to use padding.

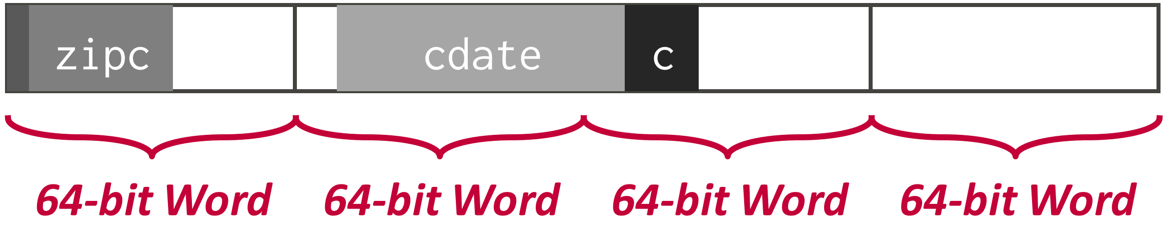
```
CREATE TABLE dj2pl (
32-bits  id INT PRIMARY KEY,
64-bits  cdate TIMESTAMP,
16-bits  color CHAR(2),
32-bits  zipcode INT
);
```



Word-alignment: Reordering

- Switch the order of attributes in the tuples' physical layout to make sure they are aligned.
  - May still have to use padding.

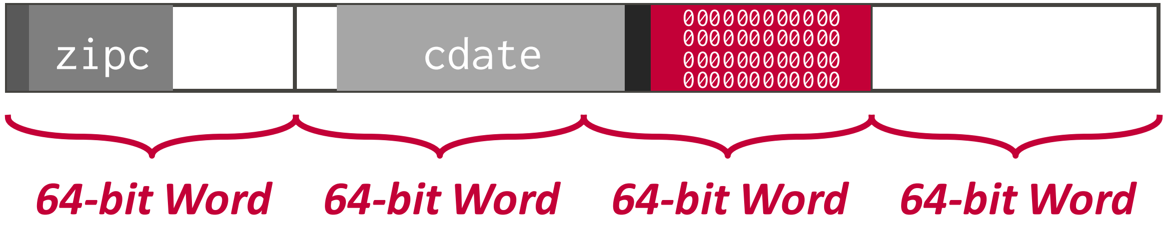
	CREATE TABLE dj2pl (
32-bits	id INT PRIMARY KEY,
64-bits	cdate TIMESTAMP,
16-bits	color CHAR(2),
32-bits	zipcode INT
	);



Word-alignment: Reordering

- Switch the order of attributes in the tuples' physical layout to make sure they are aligned.
  - May still have to use padding.

```
CREATE TABLE dj2pl (
32-bits  id INT PRIMARY KEY,
64-bits  cdate TIMESTAMP,
16-bits  color CHAR(2),
32-bits  zipcode INT
);
```



Data Representation

- INTEGER/BIGINT/SMALLINT/TINYINT

→ Same as in C/C++.

- FLOAT/REAL vs. NUMERIC/DECIMAL

→ IEEE-754 Standard / Fixed-point Decimals.

- VARCHAR/VARBINARY/TEXT/BLOB

→ Header with length, followed by data bytes **OR** pointer to another page/offset with data.

→ Need to worry about collations / sorting.

- TIME/DATE/TIMESTAMP/INTERVAL

→ 32/64-bit integer of (micro/milli)-seconds since Unix epoch (January 1st, 1970).

Data Representation

- INTEGER/BIGINT/SMALLINT/TINYINT

→ Same as in C/C++.

- FLOAT/REAL vs. NUMERIC/DECIMAL

→ IEEE-754 Standard / Fixed-point Decimals.

- VARCHAR/VARBINARY/TEXT/BLOB

→ Header with length, followed by data bytes **OR** pointer to another page/offset with data.

→ Need to worry about collations / sorting.

- TIME/DATE/TIMESTAMP/INTERVAL

→ 32/64-bit integer of (micro/milli)-seconds since Unix epoch (January 1st, 1970).

Variable Precision Numbers

SCHOOL OF DATA SCIENCE

- Inexact, variable-precision numeric type that uses the "native" C/C++ types.
- Store directly as specified by IEEE-754.

→ Example: `FLOAT`, `REAL`/`DOUBLE`

- These types are typically faster than fixed precision numbers because CPU ISA's (Xeon, Arm) have instructions / registers to support them.
- But they do not guarantee exact values...

SCHOOL OF
DATA SCIENCE

Variable Precision Numbers

Rounding Example

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    float x = 0.1;
    float y = 0.2;
    printf("x+y = %f\n", x+y);
    printf("0.3 = %f\n", 0.3);
}
```

Variable Precision Numbers

Rounding Example

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    float x = 0.1;
    float y = 0.2;
    printf("x+y = %f\n", x+y);
    printf("0.3 = %f\n", 0.3);
}
```

Output

```
x+y = 0.300000
0.3 = 0.300000
```

Variable Precision Numbers

Rounding Example

```
#include <stdio.h>

in

}
```

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    float x = 0.1;
    float y = 0.2;
    printf("x+y = %.20f\n", x+y);
    printf("0.3 = %.20f\n", 0.3);
}
```

Output

```
x+y = 0.300000
0.3 = 0.300000
```

Variable Precision Numbers

Rounding Example

```
#include <stdio.h>

in

}
```

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    float x = 0.1;
    float y = 0.2;
    printf("x+y = %.20f\n", x+y);
    printf("0.3 = %.20f\n", 0.3);
}
```

Output

```
x+y = 0.300000
0.3 = 0.300000
```

```
x+y = 0.30000001192092895508
0.3 = 0.29999999999999998890
```

Fixed Precision Numbers

- Numeric data types with (potentially) arbitrary precision and scale. Used when rounding errors are unacceptable.

→ Example: `NUMERIC`, `DECIMAL`

- Many different implementations.

→ Example: Store in an exact, variable-length binary representation with additional meta-data.

→ Can be less expensive if the DBMS does not provide arbitrary precision (e.g., decimal point can be in a different position per value).

Postgres: NUMERIC

```
typedef unsigned char NumericDigit;
typedef struct {
  int ndigits;
  int weight;
  int scale;
  int sign;
  NumericDigit *digits;
} numeric;
```

Postgres: NUMERIC

of Digits

Weight of 1st Digit

Scale Factor

Positive/Negative/NaN

Digit Storage

```
typedef unsigned char NumericDigit;
typedef struct {
  int ndigits;
  int weight;
  int scale;
  int sign;
  NumericDigit *digits;
} numeric;
```

Postgres: NUMERIC

of Digits

Weight of 1st Digit

Scale Factor

Positive/Negative/NaN

Digit Storage

```
typedef unsigned char NumericDigit;
typedef struct {
  int ndigits;
  int weight;
  int scale;
  int sign;
  NumericDigit *digits;
} numeric;
```

SCHOOL OF
DATA SCIENCE

NULL Data Types

- **Choice #1: Null Column Bitmap Header**

→ Store a bitmap in a centralized header that specifies what attributes are null.
  - This is the most common approach.

- **Choice #2: Special Values**

→ Designate a value to represent `NULL` for a data type (e.g., `INT32_MIN`).

- **Choice #3: Per Attribute Null Flag**

→ Store a flag that marks that a value is null.

→ Must use more space than just a single bit because this messes up with word alignment.

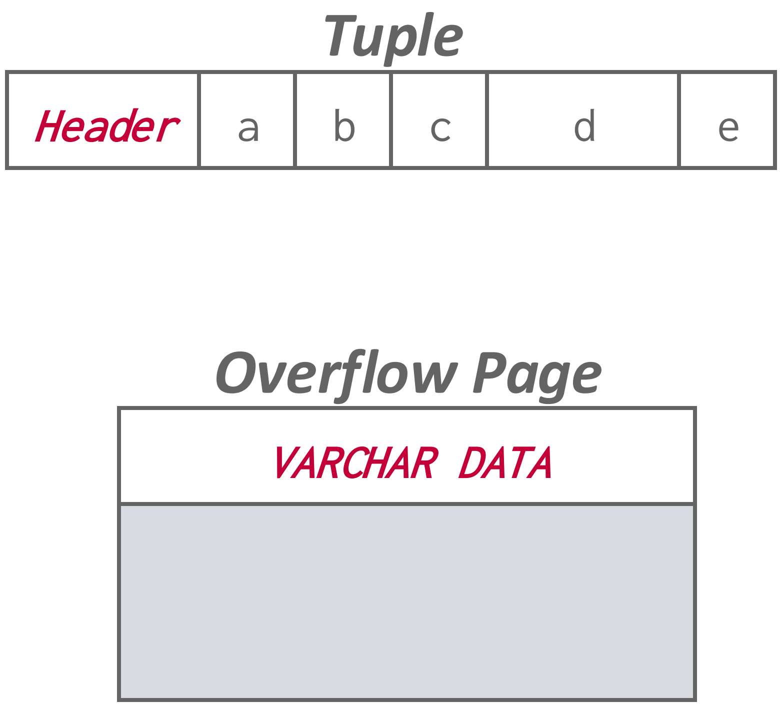
SCHOOL OF DATA SCIENCE

Large Values

- Most DBMSs don't allow a tuple to exceed the size of a single page.

- To store values that are larger than a page, the DBMS uses separate **overflow** storage pages.

→ Postgres: TOAST (>2KB)
→ MySQL: Overflow (>½ size of page)
→ SQL Server: Overflow (>size of page)



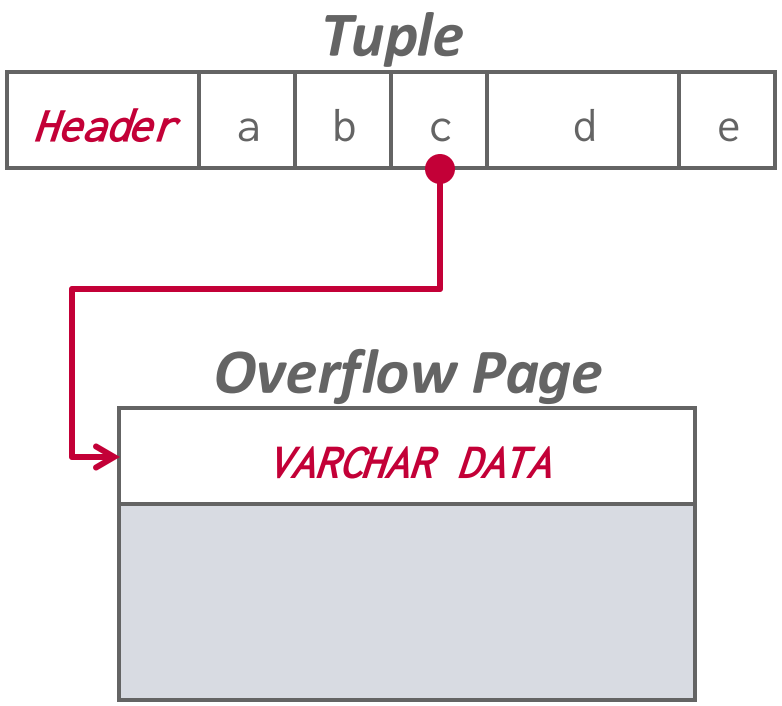
Large Values

- Most DBMSs don't allow a tuple to exceed the size of a single page.
- To store values that are larger than a page, the DBMS uses separate **overflow** storage pages.

→ Postgres: TOAST (>2KB)

→ MySQL: Overflow (>½ size of page)

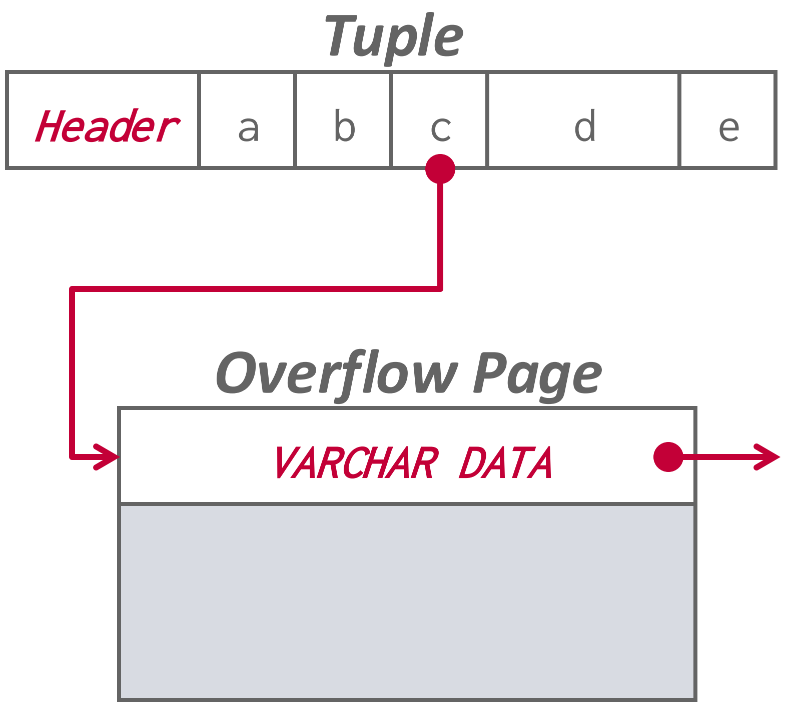
→ SQL Server: Overflow (>size of page)



Large Values

- Most DBMSs don't allow a tuple to exceed the size of a single page.
- To store values that are larger than a page, the DBMS uses separate **overflow** storage pages.

→ Postgres: TOAST (>2KB)
→ MySQL: Overflow (>½ size of page)
→ SQL Server: Overflow (>size of page)



External Value Storage

- Some systems allow you to store a large value in an external file. Treated as a `BLOB` type.

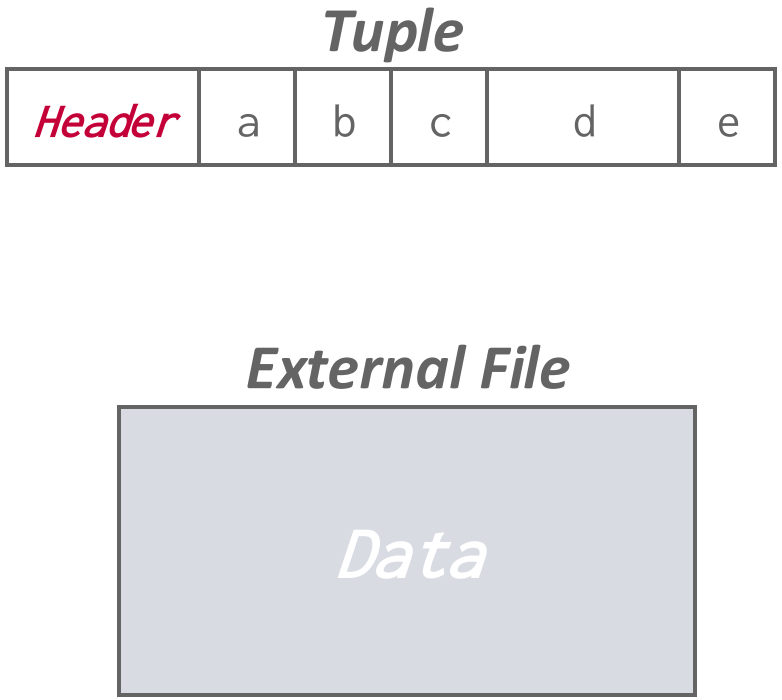
→ Oracle: `BFILE` data type

→ Microsoft: `FILESTREAM` data type

- The DBMS **cannot** manipulate the contents of an external file.

→ No durability protections.

→ No transaction protections.



External Value Storage

- Some systems allow you to store a large value in an external file. Treated as a BLOB type.

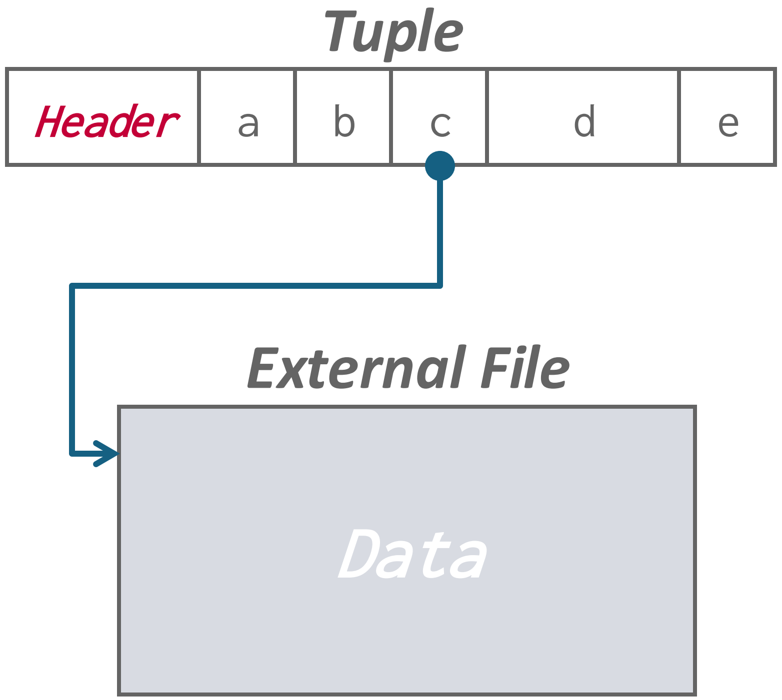
→ Oracle: BFILE data type

→ Microsoft: FILESTREAM data type

- The DBMS **cannot** manipulate the contents of an external file.

→ No durability protections.

→ No transaction protections.



External Value Storage

- Some systems allow you to store a large value in an external file. Treated as a `BLOB` type.

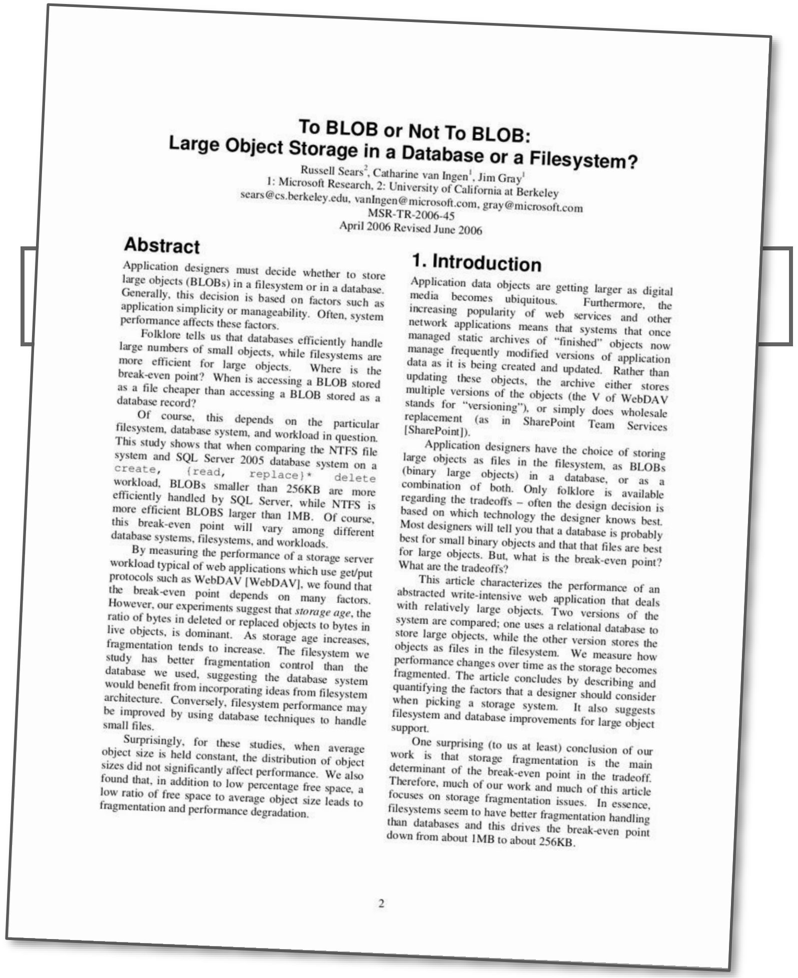
→ Oracle: `BFILE` data type

→ Microsoft: `FILESTREAM` data type

- The DBMS **<u>cannot</u>** manipulate the contents of an external file.

→ No durability protections.

→ No transaction protections.



To BLOB or Not To BLOB:
Large Object Storage in a Database or a Filesystem?

Russell Sears[2], Catharine van Ingen[1], Jim Gray[1]
1: Microsoft Research, 2: University of California at Berkeley
sears@cs.berkeley.edu, vanIngen@microsoft.com, gray@microsoft.com
MSR-TR-2006-45
April 2006 Revised June 2006

Abstract

Application designers must decide whether to store large objects (BLOBs) in a filesystem or in a database. Generally, this decision is based on factors such as application simplicity or manageability. Often, system performance affects these factors.

Folklore tells us that databases efficiently handle large numbers of small objects, while filesystems are more efficient for large objects. Where is the break-even point? When is accessing a BLOB stored as a file cheaper than accessing a BLOB stored as a database record?

Of course, this depends on the particular filesystem, database system, and workload in question. This study shows that when comparing the NTFS file system and SQL Server 2005 database system on a create, {read, replace}* delete workload, BLOBs smaller than 256KB are more efficiently handled by SQL Server, while NTFS is more efficient BLOBS larger than 1MB. Of course, this break-even point will vary among different database systems, filesystems, and workloads.

By measuring the performance of a storage server workload typical of web applications which use get/put protocols such as WebDAV [WebDAV], we found that the break-even point depends on many factors. However, our experiments suggest that *storage age*, the ratio of bytes in deleted or replaced objects to bytes in live objects, is dominant. As storage age increases, fragmentation tends to increase. The filesystem we study has better fragmentation control than the database we used, suggesting the database system would benefit from incorporating ideas from filesystem architecture. Conversely, filesystem performance may be improved by using database techniques to handle small files.

Surprisingly, for these studies, when average object size is held constant, the distribution of object sizes did not significantly affect performance. We also found that, in addition to low percentage free space, a low ratio of free space to average object size leads to fragmentation and performance degradation.

1. Introduction

Application data objects are getting larger as digital media becomes ubiquitous. Furthermore, the increasing popularity of web services and other network applications means that systems that once managed static archives of "finished" objects now manage frequently modified versions of application data as it is being created and updated. Rather than updating these objects, the archive either stores multiple versions of the objects (the V of WebDAV stands for "versioning"), or simply does wholesale replacement (as in SharePoint Team Services [SharePoint]).

Application designers have the choice of storing large objects as files in the filesystem, as BLOBs (binary large objects) in a database, or as a combination of both. Only folklore is available regarding the tradeoffs – often the design decision is based on which technology the designer knows best. Most designers will tell you that a database is probably best for small binary objects and that that files are best for large objects. But, what is the break-even point? What are the tradeoffs?

This article characterizes the performance of an abstracted write-intensive web application that deals with relatively large objects. Two versions of the system are compared; one uses a relational database to store large objects, while the other version stores the objects as files in the filesystem. We measure how performance changes over time as the storage becomes fragmented. The article concludes by describing and quantifying the factors that a designer should consider when picking a storage system. It also suggests filesystem and database improvements for large object support.

One surprising (to us at least) conclusion of our work is that storage fragmentation is the main determinant of the break-even point in the tradeoff. Therefore, much of our work and much of this article focuses on storage fragmentation issues. In essence, filesystems seem to have better fragmentation handling than databases and this drives the break-even point down from about 1MB to about 256KB.

2

Catalogs

System Catalogs

- A DBMS stores meta-data about databases in its internal catalogs.

→ Tables, columns, indexes, views

→ Users, permissions

→ Internal statistics

- Almost every DBMS stores the database's catalog inside itself (i.e., as tables).

→ Wrap object abstraction around tuples.

→ Specialized code for “bootstrapping” catalog tables.

System Catalogs

- You can query the DBMS's internal `INFORMATION_SCHEMA` catalog to get info about the database.

→ ANSI standard set of read-only views that provide info about all the tables, views, columns, and procedures in a database

- DBMSs also have non-standard shortcuts to retrieve this information.

Accessing Table Schema

- *List all the tables in the current database:*

SQL-92

```
SELECT *
  FROM INFORMATION_SCHEMA.TABLES
 WHERE table_catalog = '<db name>';
```

Postgres

```
\d;
```

MySQL

```
SHOW TABLES;
```

SQLite

```
.tables
```

Accessing Table Schema

- *List the schema of the student table:*

SQL-92
```
SELECT *
  FROM INFORMATION_SCHEMA.TABLES
 WHERE table_name = 'student'
```

Postgres
```
\d student;
```

MySQL
```
DESCRIBE student;
```

SQLite
```
.schema student
```

Conclusion

- Database is organized in pages.
- Different ways to track pages.
- Different ways to store pages.
- Different ways to store tuples.

- The storage manager is not entirely independent from the rest of the DBMS.