Lecture #03

Database Storage
Part 1
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

- **Volatile**
  - Random Access
  - Byte-Addressable

- **Non-Volatile**
  - Sequential Access
  - Block-Addressable

**Faster**
- Smaller
- Expensive

**Slower**
- Larger
- Cheaper
STORAGE HIERARCHY

- CPU Registers
- CPU Caches
- DRAM
- SSD
- HDD
- Network Storage

Faster → Smaller → Expensive → Slower → Larger → Cheaper
STORAGE HIERARCHY

- **CPU**
  - CPU Registers
  - CPU Caches

- **Memory**
  - DRAM
  - Persistent Memory
  - SSD
  - Fast Network Storage

- **Disk**
  - HDD
  - Network Storage

**Comparison:****
- Faster: CPU Registers, CPU Caches, DRAM
- Smaller: CPU Registers, CPU Caches, DRAM
- Expensive: SSD, Persistent Memory
- Slower: HDD, Network Storage
- Larger: HDD, Network Storage
- Cheaper: SSD, Persistent Memory, Fast Network Storage, Network Storage
STORAGE HIERARCHY

- **CPU**
  - CPU Registers
  - CPU Caches

- **Memory**
  - DRAM
  - Persistent Memory
  - SSD
  - Fast Network Storage

- **Disk**
  - HDD
  - Network Storage

- **Network Storage**
  - Faster
  - Smaller
  - Expensive

- **Fast Network Storage**
  - Larger
  - Cheaper
STORAGE HIERARCHY

Memory
- CPU Registers
- CPU Caches
- DRAM
- SSD
- HDD
- Network Storage

Faster
Smaller
Expensive

Slower
Larger
Cheaper

Persistent Memory

Optane DC Persistent Memory

Non-Volatile Memory Database Management Systems

Joy Arulraj
Andrew Pavlo

CMU-DB
15-445/645 (Fall 2023)
STORAGE HIERARCHY

Memory

CPU

CPU Registers

CPU Caches

DRAM

SSD

HDD

Network Storage

Faster

Smaller

Expensive

Slower

Larger

Cheaper

Persistent Memory

Fast Network Storage

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.

Intel kills the remnants of Optane memory

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

By Michael Chibbar
Staff Writer, PCWorld | JUL 29, 2022 6:19 AM PDT
## ACCESS TIMES

**Latency Numbers Every Programmer Should Know**

<table>
<thead>
<tr>
<th>Storage Type</th>
<th>Access Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Cache Ref</td>
<td>1 ns</td>
</tr>
<tr>
<td>L2 Cache Ref</td>
<td>4 ns</td>
</tr>
<tr>
<td>DRAM</td>
<td>100 ns</td>
</tr>
<tr>
<td>SSD</td>
<td>16,000 ns</td>
</tr>
<tr>
<td>HDD</td>
<td>2,000,000 ns</td>
</tr>
<tr>
<td>Network Storage</td>
<td>~50,000,000 ns</td>
</tr>
<tr>
<td>Tape Archives</td>
<td>1,000,000,000 ns</td>
</tr>
</tbody>
</table>

Source: Colin Scott
## ACCESS TIMES

### Latency Numbers Every Programmer Should Know

<table>
<thead>
<tr>
<th>Access Time</th>
<th>Storage Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ns</td>
<td>L1 Cache Ref</td>
</tr>
<tr>
<td>4 ns</td>
<td>L2 Cache Ref</td>
</tr>
<tr>
<td>100 ns</td>
<td>DRAM</td>
</tr>
<tr>
<td>16,000 ns</td>
<td>SSD</td>
</tr>
<tr>
<td>2,000,000 ns</td>
<td>HDD</td>
</tr>
<tr>
<td>~50,000,000 ns</td>
<td>Network Storage</td>
</tr>
<tr>
<td>1,000,000,000 ns</td>
<td>Tape Archives</td>
</tr>
<tr>
<td>1 sec</td>
<td></td>
</tr>
<tr>
<td>4 sec</td>
<td></td>
</tr>
<tr>
<td>100 sec</td>
<td></td>
</tr>
<tr>
<td>4.4 hours</td>
<td></td>
</tr>
<tr>
<td>3.3 weeks</td>
<td></td>
</tr>
<tr>
<td>1.5 years</td>
<td></td>
</tr>
<tr>
<td>31.7 years</td>
<td></td>
</tr>
</tbody>
</table>

Source: Colin Scott
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

Get Page #2

Memory

Buffer Pool

Directory

Header

Pages

Database File

Directory

Header

1

2

3

4

5

...
DISK-ORIENTED DBMS
DISK-ORIENTED DBMS

Memory

Disk

Buffer Pool

Directory

Header

2

Page

Database File

Directory

Header

1

2

3

4

5

Pages

Get Page #2

Execution Engine

Pointer to Page #2

Interpret Page #2 layout

Update Page #2

Get Page #2

Update Page #2
DISK-ORIENTED DBMS

Lecture #6

Get Page #2

Lectures #12-13

Execution Engine

Pointer to Page #2

Interpret Page #2 layout

Update Page #2

Lectures #3-5

Memory

Disk

Buffer Pool

Database File

Directory

Header

1

2

3

4

5

Pages
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 (\texttt{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.
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?

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 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.
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.
TODAY'S AGENDA

File Storage
Page Layout
Tuple Layout
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.
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.
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.
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.
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.
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.
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
Every page contains a **header** of metadata 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).
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**
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**
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.

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num Tuples = 3</td>
</tr>
<tr>
<td>Tuple #1</td>
</tr>
<tr>
<td>Tuple #2</td>
</tr>
<tr>
<td>Tuple #3</td>
</tr>
</tbody>
</table>
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?

<table>
<thead>
<tr>
<th>Page</th>
<th>Num Tuples = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tuple #1</td>
</tr>
<tr>
<td></td>
<td>Tuple #3</td>
</tr>
</tbody>
</table>
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?

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num Tuples = 3</td>
</tr>
<tr>
<td>Tuple #1</td>
</tr>
<tr>
<td>Tuple #4</td>
</tr>
<tr>
<td>Tuple #3</td>
</tr>
</tbody>
</table>
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?
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.
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.
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.
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.
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.
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.
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.
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.

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

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

$\rightarrow$ Potentially reduces the amount of I/O for common workload patterns.

$\rightarrow$ Can make updates more expensive.

\begin{verbatim}
SELECT * FROM foo JOIN bar
ON foo.a = bar.a;
\end{verbatim}
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