Lecture #03

Database Storage

Part 1
ADMINISTRIVIA

**Homework #1** is due February 2\textsuperscript{nd} @ 11:59pm

**Project #0** is due January 28\textsuperscript{th} @ 11:59pm

**Project #1** will be released on February 5\textsuperscript{th}
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

Application

Query Planning
Operator Execution
Access Methods
Buffer Pool Manager
Disk Manager

SQL
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

- **CPU Registers**
  - Volatile
  - Random Access
  - Byte-Addressable

- **CPU Caches**
  - Faster
  - Smaller
  - Expensive

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

- **SSD**
  - Slower
  - Larger
  - Cheaper

- **HDD**

- **Network Storage**
STORAGE HIERARCHY

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

- **Memory**
  - DRAM

- **Disk**
  - SSD
  - HDD

- **Network Storage**

Properties:
- **Faster**
- **Smaller**
- **Expensive**
- **Slower**
- **Larger**
- **Cheaper**
STORAGE HIERARCHY

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

- Memory
- Disk

Faster, Smaller, Expensive

- Network Storage: Faster, Larger, Cheaper
• 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.
Every programmer must know these numbers.

Jeff Dean

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

Lecture #6

Get Page #2

Lectures #13-14

Execution Engine

Pointer to Page #2

Interpret Page #2 layout

Update Page #2

Lecture #6

Lectures #3-5

Pages

Memory

Buffer Pool

Disk

Database 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 (\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?

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…
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.
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.
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?

Andrew Crotty
Carnegie Mellon University
andrew@cmu.edu

Viktor Lets
University of Erlangen-Nuremberg
viktor.lets@fau.de

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

ABSTRACT

Memory-mapped (map) 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 were stored in memory. OS-transparent behind-the-scenes pages updates occur when the program references them and automatically commits pages to memory as the system demands. Most OSes provide an interface for running a program on a disk file. Generating the disk file is easy and almost always does the right thing. The OS does a better job of maintaining the disk file than the application, which means that the OS is your friend.

→ Flushing dirty pages to disk in the correct order.

→ 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?

https://db.cs.cmu.edu/mmap-cldr2022
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.
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.

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

**Default DB Page Sizes**

- 4KB: SQLite, ORACLE
- 8KB: IBM DB2, RocksDB
- 16KB: Microsoft SQL Server, PostgreSQL
- 32KB: 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.
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.
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.
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.
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 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).
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**
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?

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num Tuples = 2</td>
</tr>
<tr>
<td>Tuple #1</td>
</tr>
<tr>
<td>Tuple #4</td>
</tr>
<tr>
<td>Tuple #3</td>
</tr>
</tbody>
</table>
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.
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.
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.
TODAY’S AGENDA

File Storage

Page 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.
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 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.

```
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.

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.

```
SELECT * FROM foo JOIN bar
ON foo.a = bar.a;
```
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
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.
NEW FORMATS: THE SEARCH CONTINUES ...

Bonus

The gift of Moore’s Law is ending

Notes

Data growth is exponential.

Historical hardware performance doubled exponentially too, providing a “way out.”

The free lunch is over.

The current solution of “scaling out” is not sustainable. Need to “scale-in” to the hardware layers.
A POTENTIAL SOLUTION: DO MORE WITH LESS

There is untapped parallelism in most computing substrate, and lots of it. Can we exploit this for data processing?
Consider the list of numbers below:

- 6708, 6881, 8554, 1878, 5362, 1930, 5677, 6650, 5149, 4716

Task: Identify numbers below 2000.

- The usual approach is to scan the list left to right, and check if it's under 2000.

How much data must be examined for this task?

- This simple scan is often the most data-hungry operation in analytic queries.
INTUITION

Data: 6708, 6881, 8554, 1878, 5362, 1930, 5677, 6650, 5149, 4716
Task: Identify numbers below 2000

Bonus
Data: 6708, 6818, 8554, 1878, 5362, 1930, 5677, 6650, 5149, 4716
Task: Identify numbers below 2000

| 6 | 6 | 8 | 1 | 5 | 1 | 5 | 6 | 5 | 4 | 7 | 8 | 5 | 8 | 3 | 9 | 6 | 6 | 1 | 7 | 0 | 8 | 5 | 7 | 6 | 3 | 7 | 5 | 4 | 1 | 8 | 1 | 4 | 8 | 2 | 0 | 7 | 0 | 9 | 6 |

10 cells
BITWEAVING/V

First batch of Processor Words
(batch size = code size in bits)

Column 1,
Value: 101010

MSB of the first column
MSB of the second column
MSB of the 64th column

4-bit column

4th bit/LSB of the first column

Details: SIGMOD’13, VLDB’14, SIGMOD’15, SIGMOD’24 (to appear)
**BITWEAVING/V: EARLY PRUNING**

<table>
<thead>
<tr>
<th>Column Codes:</th>
<th>Constant</th>
<th>Predicate</th>
<th>Result Bit Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 12 3 6 9 7 1 0</td>
<td>5</td>
<td><code>colValue &lt; 5</code></td>
<td>0 0 1 0 0 0 1 1</td>
</tr>
<tr>
<td>1 1 0 0 1 0 0 0</td>
<td>0</td>
<td>✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗</td>
<td></td>
</tr>
<tr>
<td>0 1 0 1 0 1 0 0</td>
<td>1</td>
<td>✗ ✗ ✓ ✗ ✗ ✓ ✗ ✓</td>
<td></td>
</tr>
<tr>
<td>1 0 1 1 0 1 0 0</td>
<td>0</td>
<td>✓ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗</td>
<td></td>
</tr>
<tr>
<td>0 0 1 0 1 1 1 0</td>
<td>1</td>
<td>✗ ✗ ✗ ✗ ✗ ✗ ✗ ✓ ✓</td>
<td></td>
</tr>
</tbody>
</table>

Early Pruning: skip the last check
The algorithmic advantage arises from the intrinsic properties of the encoding, leading to fewer bits that need to be examined.

Early pruning probability: ~80% at bit position 8

\[ P(b) = (1 - \left(\frac{1}{2}\right)^b)^w \]
Intel Xeon X5650; 24GB DRAM; COUNT(*) query on synthetic data with 1B tuples.
A true co-design approach with changes on both the algorithm and hardware sides.

Past work has largely tried to “fit” existing algorithms to new hardware.
NEXT CLASS

Log-Structured Storage
Index-Organized Storage
Value Representation
Catalogs