

# Lecture 4: Disk Space Management

CREATING THE NEXT®

# Administrivia

---

- Collaboration allowed for programming assignments
- Exercise sheets and exams will be individual tasks
- Assignment 1 is due on September 8th @ 11:59pm

# Today's Agenda

---

Recap

Layered Architecture

Hardware Properties

Disk-Oriented DBMS

File Storage

Page Layout

Tuple Layout

# Recap

# List of SQL Features

---

- Aggregations + Group By
- String / Date / Time Operations
- Output Control + Redirection
- Nested Queries
- Join
- Common Table Expressions
- Window Functions

# Window Functions

---

- **Task:** Get the name of the students with the second highest grade for each course.

```
SELECT cid, sid, grade, rank FROM (  
  SELECT *, RANK()  
    OVER (PARTITION BY cid ORDER BY grade ASC) AS rank  
  FROM enrolled  
) AS ranking  
WHERE ranking.rank = 2 --- Update rank
```

cid	sid	grade	rank
2	4	C	2

# Common Table Expressions

---

- **Task:** Find students record with the highest id that is enrolled in at least one course.

```
WITH cteSource (maxId) AS (  
    SELECT MAX(sid) FROM enrolled  
)  
SELECT name FROM students, cteSource  
WHERE students.sid = cteSource.maxId
```

## Types of Join: Lateral Join

---

- **Task:** List the names of students with hobbies (get student name once for each occurrence of their hobby).

```
SELECT name  
FROM students, LATERAL (SELECT id FROM hobbies  
WHERE students.id = hobbies.id) ss;
```

—————  
**name**  
—————

Maria

Maria

Rahul

Peter  
—————



# Layered Architecture

# Overview

---

- We now understand what a database looks like at a **logical** level and how to write queries to read/write data from it (*i.e.*, **physical** level).
- We will next learn how to build software that manages a database.

# Anatomy of a Database System [Monologue]

---

- Process Manager
  - ▶ Manages client connections
- Query Processor
  - ▶ Parse, plan and execute queries on top of storage manager
- Transactional Storage Manager
  - ▶ Knits together buffer management, concurrency control, logging and recovery
- Shared Utilities
  - ▶ Manage hardware resources across threads

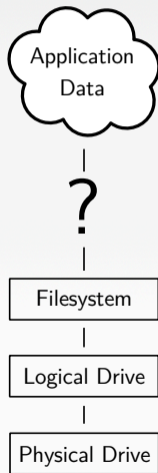
## Anatomy of a Database System [Monologue] (2)

---

- Process Manager
  - ▶ Connection Manager + Admission Control
- Query Processor
  - ▶ Query Parser
  - ▶ Query Optimizer (*a.k.a.*, Query Planner)
  - ▶ Query Executor
- Transactional Storage Manager
  - ▶ Lock Manager
  - ▶ Access Methods (*a.k.a.*, Indexes)
  - ▶ Buffer Pool Manager
  - ▶ Log Manager
- Shared Utilities
  - ▶ Memory, Disk, and Networking Manager

# The Problem

---



# Requirements

---

There are different classes of requirements:

- Data Independence
  - ▶ application logic must be shielded from physical storage implementation details
  - ▶ physical storage can be reorganized
  - ▶ hardware can be changed
- Scalability
  - ▶ must scale to (nearly) arbitrary data size
  - ▶ efficiently access to individual tuples
  - ▶ efficiently update an arbitrary subset of tuples
- Reliability
  - ▶ data must never be lost
  - ▶ must cope with hardware and software failures
- ...

# Layered Architecture

---

- implementing all these requirements on “bare metal” is hard
- and not desirable
- a DBMS must be maintainable and extensible

Instead: use a layered architecture

- the DBMS logic is split into levels of functionality
- each level is implemented by a specific layer
- each layer interacts only with the next lower layer
- simplifies and modularizes the code

# A Simple Layered Architecture

---

## Purpose

query translation  
and optimization

managing records  
and access paths

DB buffer and  
hardware interface

query layer

access layer

storage layer



## Access Granularity

declarative queries  
sets of records

records

page



## A Simple Layered Architecture (2)

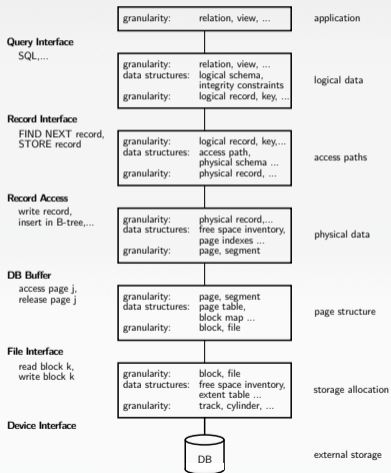
---

- layers can be characterized by the data items they manipulate
- lower layer offers functionality for the next higher level
- keeps the complexity of individual layers reasonable
- rough structure: physical → low level → high level

This is a reasonable architecture, but simplified.

A more detailed architecture is needed for a complete DBMS.

# A More Detailed Architecture



## A More Detailed Architecture (2)

---

A few pieces are still missing:

- transaction isolation
- recovery

but otherwise it is a reasonable architecture.

Some system deviate slightly from this classical architecture

- many DBMSs nowadays delegate disk access to the OS
- some DBMSs delegate **buffer management** to the OS (tricky, though)
- a few DBMSs allow for direct logical record access
- ...

# Hardware Properties

# Impact of Hardware

---

Must take hardware properties into account when designing a storage system.

For a long time dominated by **Moore's Law**:

*The number of transistors on a chip doubles every 18 months.*

Indirectly drove a number of other parameters:

- main memory size
- CPU speed
  - ▶ no longer true!
- HDD capacity
  - ▶ start getting problematic, too. density is very high
  - ▶ only capacity, not access time

# Memory Hierarchy

---

capacity  
latency

bytes  
1ns

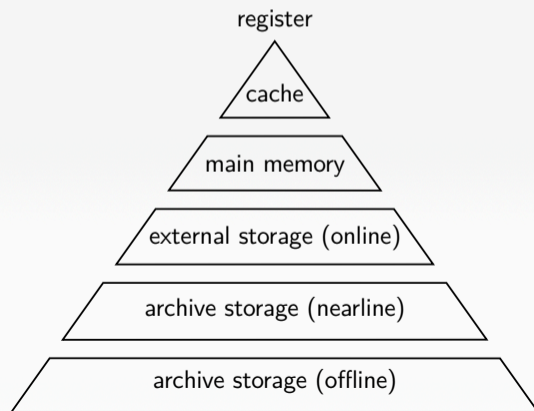
K-M bytes  
<10ns

G bytes  
<100ns

T bytes  
ms

T bytes  
sec

T-P bytes  
sec-min



## Memory Hierarchy (2)

---

There are huge gaps between hierarchy levels

- traditionally, main memory vs. disk is most important
- but memory vs. cache etc. also relevant

The DBMS must aim to maximize locality.

# Hard Disk Access

---

Hard Disks are still the dominant external storage:

- rotating platters, mechanical effects
- transfer rate: ca. 150MB/s
- seek time ca. 3ms
- huge imbalance in random vs. sequential I/O!



## Hard Disk Access (2)

---

The DBMS must take these effects into account

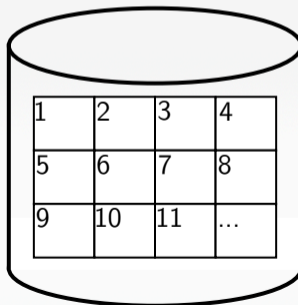
- sequential access is much more efficient
- traditional DBMSs are designed to maximize sequential access
- gap is growing instead of shrinking
- even SSDs are slightly asymmetric (and have other problems)
- DBMSs try to reduce number of writes to random pages by organizing data in **contiguous blocks**.
- Allocating multiple pages at the same time is called a **segment**

## Hard Disk Access (3)

---

Techniques to speed up disk access:

- do not move the head for every single tuple
- instead, load larger chunks. typical granularity: one **page**
- page size varies. traditionally 4KB, nowadays often 16K and more (trade-off)



## Hard Disk Access (4)

---

The page structure is very prominent within the DBMS

- granularity of I/O
- granularity of buffering/memory management
- granularity of recovery

Page is still too small to hide random I/O though

- sequential page access is important
- DBMSs use read-ahead techniques
- asynchronous write-back

# Database System Architectures

---

## Storage Management

### Disk-Centric Database System

- The DBMS assumes that the primary storage location of the database is HDD.

### Memory-Centric Database System (MMDB)

- The DBMS assumes that the primary storage location of the database is DRAM.

## Buffer Management

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

## Access Times

---

Access Time	Hardware	Scaled Time
0.5 ns	L1 Cache	0.5 sec
7 ns	L2 Cache	7 sec
100 ns	DRAM	100 sec
350 ns	NVM	6 min
150 us	SSD	1.7 days
10 ms	HDD	16.5 weeks
30 ms	Network Storage	11.4 months
1 s	Tape Archives	31.7 years

Source: [Latency numbers every programmer should know](#)

# Disk-Oriented DBMS

# 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.

# Disk-Oriented DBMS

---

Query execution engine → Storage Manager: Get Page 2

**Memory | Buffer Pool**

Page Directory	-	-	-
----------------	---	---	---

**Disk | Database File**

Page Directory	8	5	1	4	7	3	2	9	6
----------------	---	---	---	---	---	---	---	---	---



# Disk-Oriented DBMS

---

- Each page has a **header** with the page's metadata (*e.g.*, page number, free space bitmap)
- Query execution engine gets pointer to page 2
  - ▶ Interprets the contents of page 2 using the header
- **Page directory** is typically implemented as a hash table
  - ▶ page number → buffer pool slot
  - ▶ page number → file block
- Page migration between disk and memory is known as buffer management

## Why not use the OS?

---

- One can use memory mapping (mmap) to store the contents of a file into a process' address space.
- The OS is responsible for moving data for moving the files' pages in and out of memory.

### Problems

- 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.

## Why not use the OS?

---

- There are some solutions to this problem:
  - ▶ **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.
- Database systems using mmap
  - ▶ Full Usage: MonetDB, LMDB, *e.t.c.*
  - ▶ Partial Usage: mongoDB, MemSQL, *e.t.c.*

# Why not use the OS?

---

- DBMS (almost) always wants to control things itself and can do a better job at it.
  - ▶ Flushing dirty pages to disk in the correct order.
  - ▶ Specialized prefetching.
  - ▶ Buffer replacement policy.
  - ▶ Thread/process scheduling.

# Storage Management

---

- File Storage
- Page Layout
- Tuple Layout

# File Storage

# File Storage

---

- The DBMS stores a database as one or more files on disk.
  - ▶ The OS doesn't know anything about the contents of these files.
- Early systems in the 1980s used custom filesystems on raw storage.
  - ▶ Some "enterprise" DBMSs still support this.
  - ▶ Most newer DBMSs do not roll their own filesystem

# Storage Manager

---

- The **storage manager** is responsible for maintaining a database's files.
  - ▶ Some do their own **scheduling** of I/O operations to improve spatial and temporal locality of pages.
- It organizes the files as a collection of pages.
  - ▶ Tracks data being read from and written to pages.
  - ▶ Tracks the available free space.



# 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. Why?
- Each page is given a unique identifier.
  - ▶ The DBMS uses an **indirection layer** to map page ids to physical locations.
  - ▶ This is implemented as a page directory table.

# Database Pages

---

- There are three different notions of "pages" in a DBMS:
  - ▶ Hardware Page (usually 4 KB)
  - ▶ OS Page (usually 4 KB)
  - ▶ Database Page (512 B – 16 KB)
- By hardware page, we mean at what level the device can guarantee a "failsafe write".

# Page Storage Architectures

---

- Different DBMSs manage pages in files on disk in different ways.
  - ▶ Heap File Organization
  - ▶ Sequential / Sorted File Organization
  - ▶ Hashing File Organization
- At this point in the hierarchy we don't need to know anything about what is inside of the pages.

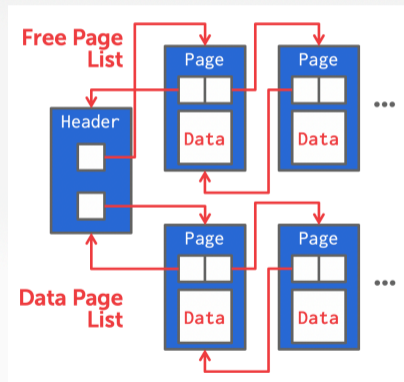
# Database Heap

---

- A **heap file** is an unordered collection of pages where tuples are stored in random order.
  - ▶ Create / Get / Write / Delete Page
  - ▶ Must also support iterating over all pages.
- Need meta-data to keep track of what pages exist and which ones have free space.
- Two ways to represent a heap file:
  - ▶ Linked List
  - ▶ Page Directory

## Heap File Organization: Linked List

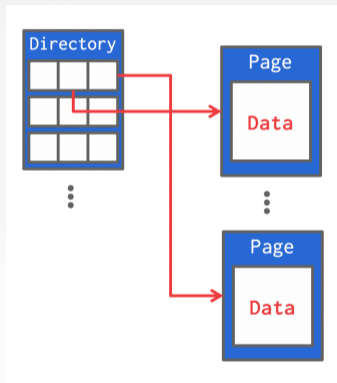
- Maintain a **header page** at the beginning of the file that stores two pointers:
  - ▶ HEAD of the **free page list**.
  - ▶ HEAD of the **data page list**.
- Each page keeps track of the number of free slots in itself.



## Heap File Organization: Page Directory

---

- The DBMS maintains special pages that tracks the location of data pages in the database files.
- The directory also records the number of free slots per page.
- The DBMS has to make sure that the directory pages are in sync with the data pages.



# Page Layout

# Page Header

---

- Every page contains a header of meta-data about the page's contents.
  - ▶ Page Size
  - ▶ Checksum
  - ▶ DBMS Version
  - ▶ Transaction Visibility
  - ▶ Compression Information
- Some systems require pages to be **self-contained** (e.g., Oracle).



# Page Layout

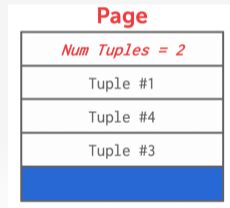
---

- For any page storage architecture, we now need to understand how to organize the data stored inside of the page.
  - ▶ We are still assuming that we are only storing tuples.
- Two approaches:
  - ▶ Tuple-oriented
  - ▶ Log-structured

# Tuple Storage

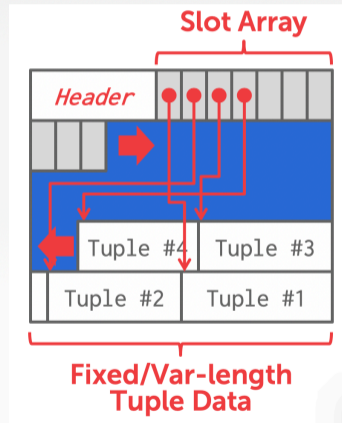
---

- 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 page layout scheme is called slotted pages.
- The **slot array** maps "slots" to the tuples' starting position offsets.
- The header keeps track of:
  - ▶ The number of used slots
  - ▶ The offset of the starting location of the last slot used.



# 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 keeping track of NULL values.
- We do not need to store meta-data about the schema. Why?

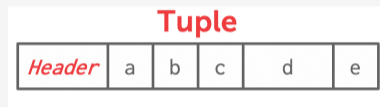


# Tuple Data

---

- Attributes are typically stored in the order that you specify them when you create the table.
- This is done for software engineering reasons.

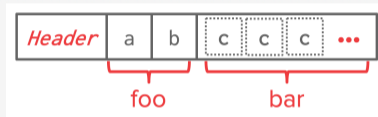
```
CREATE TABLE foo (  
  ^Ia INT PRIMARY KEY,  
  ^Ib INT NOT NULL,  
  ^Ic INT,  
  ^Id DOUBLE,  
  ^Ie FLOAT  
);
```



## Denormalized Tuple Data

- 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.
  - ▶ IBM System R did this in the 1970s.
  - ▶ Several NoSQL DBMSs do this as well.

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





# Tuple IDs

---

- The DBMS needs a way to keep track of individual tuples.
- Each tuple is assigned a unique record identifier.
  - ▶ Most common: page\_id + offset/slot
  - ▶ Can also contain file location info.
- An application **cannot** rely on these ids to mean anything.
- Examples
  - ▶ PostgreSQL: CTID (6-bytes)
  - ▶ SQLite: ROWID (10-bytes)
  - ▶ Oracle: ROWID (8-bytes)

## Conclusion

---

- Database systems have a layered architecture.
- Design of database system components affected by hardware properties.
- Database is physically organized as a collection of pages on disk.
- Different ways to manage pages and tuples.

# Next Class

---

- Memory Management