Lecture 1 - Architecture

Intro to CS286 Logistics

- Welcome!
- This is a research-based grad class. Focus is reading and discussing papers and doing research. Primarily VLDB, SIGMOD tradition. We will cover old papers as well as new ones.
- Website link: https://cs286berkeley.github.io/sp26/
- Grading logistics:
  - Project: 75%
    - Do something cool - write about it
    - Basically "paper-worthy" at a data management venue
  - Summaries: 10%
    - No submission expected today - but from next class onwards we will expect you to submit a brief summary of what you learned.
    - Lightly graded; read all assigned papers per class, and pick one for which you share something interesting you took away (i.e., a short para, even 3-4 lines OK!)—not an LLM-generated summary.
    - Google form submit anytime before class
    - Skip 4
  - Participation: 15%
    - As of now, I plan to teach throughout (with maybe some guest lectures)
    - Welcome any and all questions
    - Expect that you attend and participate. OK with me to skip a few lectures (no notice necessary) - but if you skip a lot, defeats the purpose of the class
- Prerequisites: happy to discuss after class on a case-by-case basis.
  - I will generally assume that you have background corresponding to an undergraduate database class
    - So if the terms "two phase locking", "write ahead log", "predicate pushdown"
- Course goals:
  - be comfortable with reading and discussing papers in data management
  - lead a research project in data management
  - study for the prelim exam in data management
- If you've never read papers before, there are some references on the course website that you can leverage
  - Bottomline: don't expect to understand everything
  - First quick pass "skim" to get the lay of the land, abstract, intro, skim of rest
    - What type of paper is it: theory, systems, survey
    - What is the main contribution? How is it evaluated
    - For papers in the wild, you can choose to drop the paper here if it isn't interesting
  - Then, deeper dive into the paper - read carefully, but skip over things that don't make sense to you as long as it's not essential to understand
    - Most course papers you should get to this stage
  - Finally, dig even deeper - here only if you're trying to reimplement/extend. You should understand everything
  - Generally, it is quite hard to follow research papers in the beginning - will look like "greek"! I promise you it gets better.
- How the class will go: I've never taught this class
  - I will use the whiteboard to present some stuff. You can interrupt anytime. I don't have all the answers. We will try to discover stuff together.
  - The goal is to have a conversation, not a lecture
  - For those of you who are new to a research class, nobody has all the answers - certainly not the instructor - expect that there will be lots of open questions

- We will not cover every detail of every paper, but we will try to discuss what we can.
- Any Q?

Intro to CS286 Material

A bit of course history

- Class last taught in 2024 Spring, and then before that in 2020. Not a lot of database faculty so we may not be able to offer it very often.
- This class will be on data systems, broadly, but at least in the early part will focus primarily on databases
  - Classical title "implementation of data base systems"
    - This dates back to Mike Stonebraker, who developed the course originally
  - Most work in the databases community now is focused on systems that use database systems principles, but may not be databases
  - For example, a "cloud data warehouse", "key-value store", "document store", a "streaming engine", "a business intelligence tool"...
  - We may use the terms interchangeably
- Primary data systems conferences: VLDB, SIGMOD, also ICDE, CIDR, PODS
- The class had a "textbook" called "readings in database systems" - mostly a collection of papers with some commentary. Is fairly out of date - last (fifth) edition is from 2015.
  - also called the redbook

So, a database

- Q: What are the main properties of a database system? What is it meant to do?
  - Manage and maintain data over time
  - Be "user friendly": declarative, high level queries translated, optimized, executed automatically
  - Support processing at scale, efficiently
  - Support many users concurrently
  - Be reliable in the face of failures
  - Manage access securely
- Central idea: **support access, processing, and management of data at scale**
- Hence the term *data management* for the field
- The database systems field, since it centers on *data management*, rather than a particular technique, or a particular approach, ends up intersecting with lots of other fields:
  - Theory - subfield of database theory "PODS" - intersecting both with algorithms and logic
  - PL - declarative query languages, other data systems have other DSLs
  - OS, dist systems, NT - the database is a complex software system that overlaps a lot with the capabilities of operating systems, and also requires networking capabilites when run on data centers/clusters
  - Arch - various characteristics can have a huge impact on database performance (see DAMON workshop at SIGMOD)
  - HCI - human factors, interfaces
  - Contributions therefore can vary a lot
- Within the field of systems software, databases are unique:
  - complex, mission-critical, but specialized software systems
  - decades of academic/industry research
  - among the earliest widely deployed online server systems
  - Due to this history, databases pioneered techniques that became later commonplace in distributed systems, operating systems, and large scale networked services

- Database systems underpin most of the world's app infrastructure: from the paper "*e-commerce, medical records, billing, human resources, payroll, customer relationship management and supply chain management*" - you will often hear the term "source of truth" or "system of record".
- In addition, database systems also power what is known as business intelligence, or "analytics": making sense of data at scale - across industries and sectors

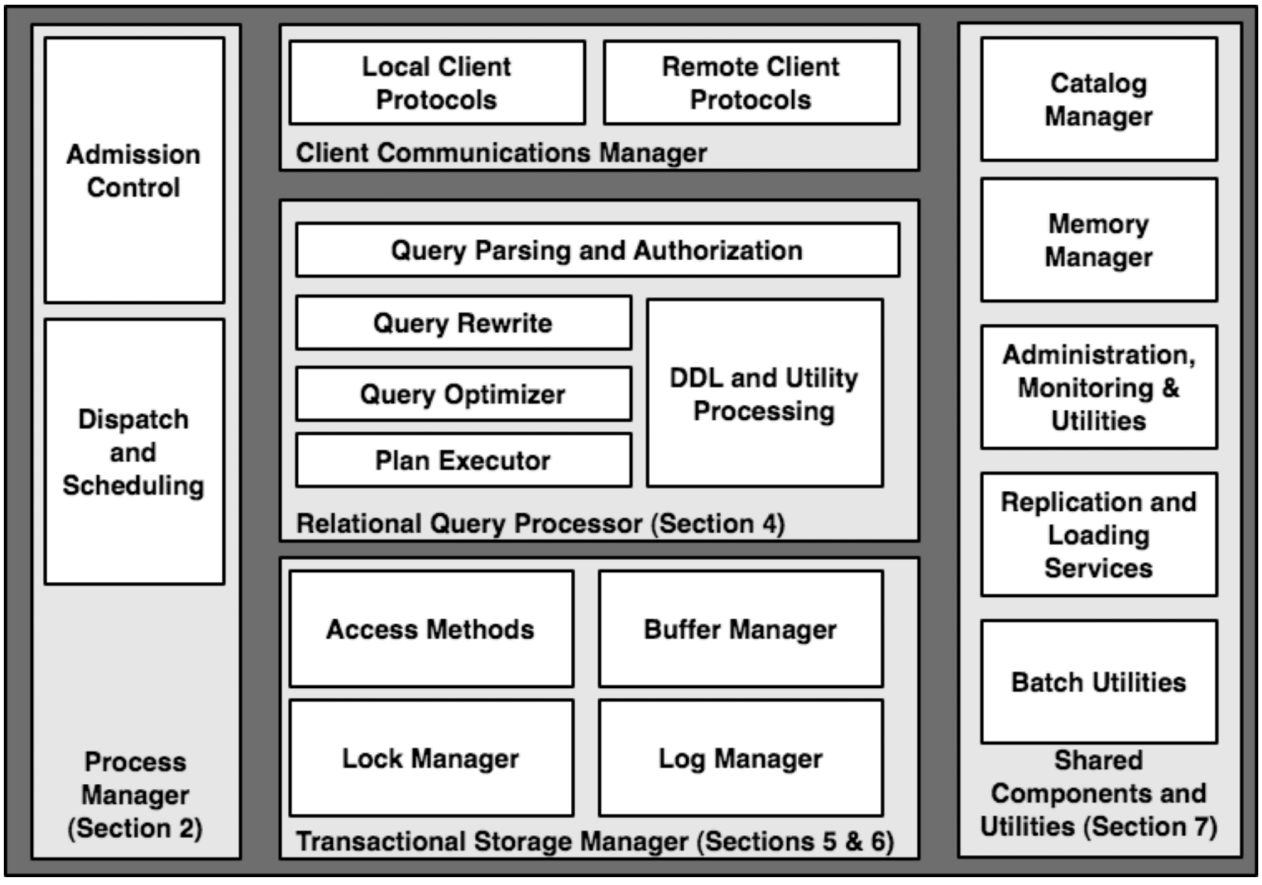
DB Pre-History

- Pre-codd: Codasyl, IMS: networked, hierarchical models
- Codd: relational model - theoretical construct
- Ingres (Berkeley) and System R (IBM) operationalized. Oracle followed suit (but won the market)
- 80s onwards - mainstream relational databases everywhere
  - major movements of that era: parallel/distributed DBs

The present paper

- Q: Who read this?
- More a survey of practical aspects of designing databases that doesn't usually make its way to papers; most papers focus on more algorithmic aspects
- Gem of a paper - so much practical information about database systems that is hard to "teach"
- BUT: is also a survey of most of 286! We will instead read original papers for most of this instead.
- Highly recommend you read again once you're done with 286.
- Goal today: get through various DB components, but hope to convey a picture of the overall architecture

Life of a query



- (In class, draw just four boxes: process mgr, query processor, storage manager, utilities)
- Most DB classes don't talk through what happens when queries arrive

1. User (e.g., someone making a purchase) establishes a connection with the DB - happens through DB IDE or through apps via some protocol (OpenDBC/JDBC), or an abstraction (ORM) that translates to DB queries
2. Once established, the process manager starts accepting queries from the user; some may not even be admitted. Query assigned to a DBMS worker
3. Query is handed to the query processor -> checks authorization, parsed, rewritten compiles to query plan comprising one or more operators (joins, aggregation, etc)
    1. During auth, checks if you're even allowed to do the action (eg SELECT, INSERT) on the objects (tables, view) in question
        1. Metadata is stored in the "Catalog" - just more tables! Dogfooding
    2. Query parsed -> rewritten (views inlined, unnesting)
    3. QO looks at search space of legal plans, uses cost estimates to pick cheapest - exponential search
    4. Plan Executor: A DSL of physical operators
4. The operator implementations make data access requests, responsibility of the storage manager
    - responsible for "access methods" (seq scan, indexes), data fetched in is stored in the buffer. "leaves" of query plan
    - since we want ACID guarantees, will need some mechanism to coordinate low-level data operations across queries - one such mechanism is locks.
        - will need to lock both tuples and indexing structures (e.g B+trees)
    - and also log actions if we're planning to edit data on disk (write ahead log)
5. Results returned - sometimes not all of it so that it doesn't overwhelm the client. The current location of the client is called the cursor

- Modern nuances: some of these are now network calls, e.g., authentication may be handled by a separate service; metadata catalog may be a separate service, logging and recovery may be a separate service, etc.

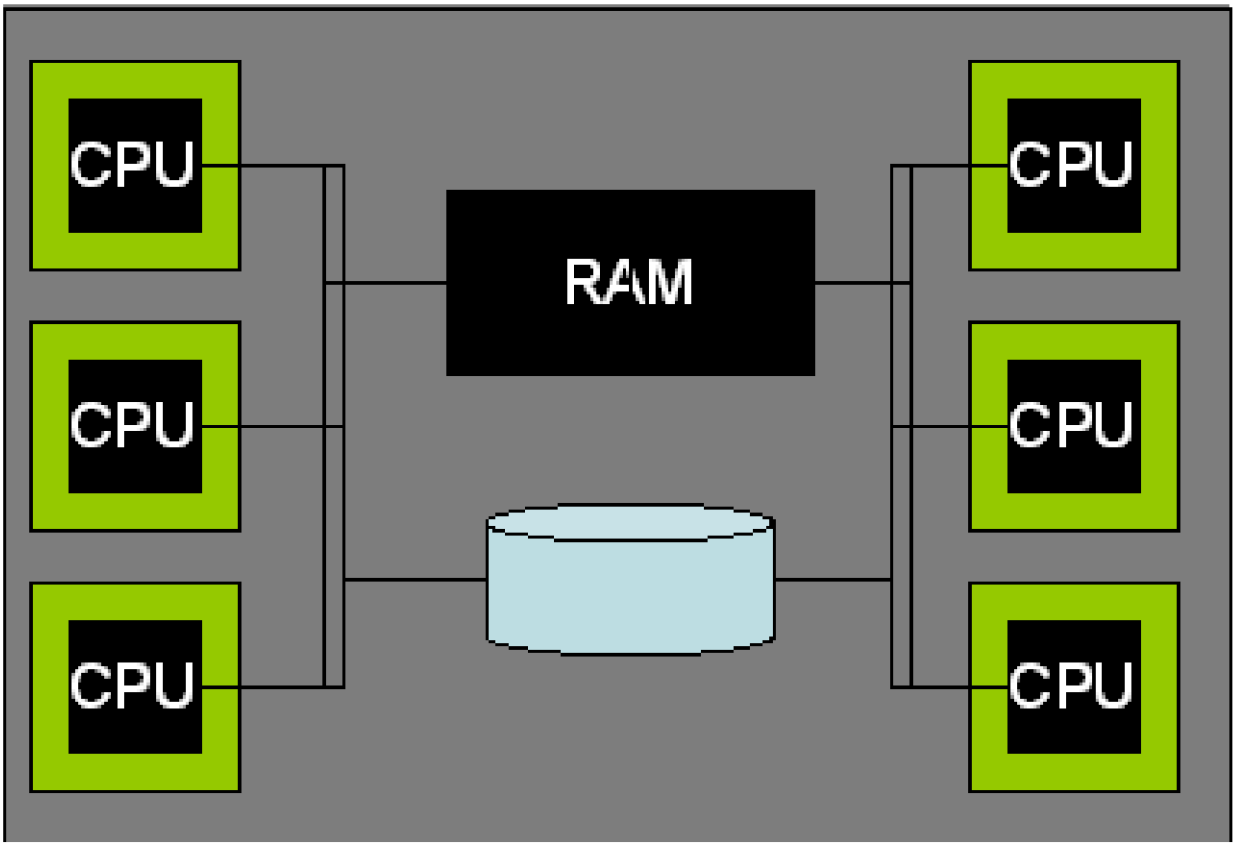
1) Process Manager

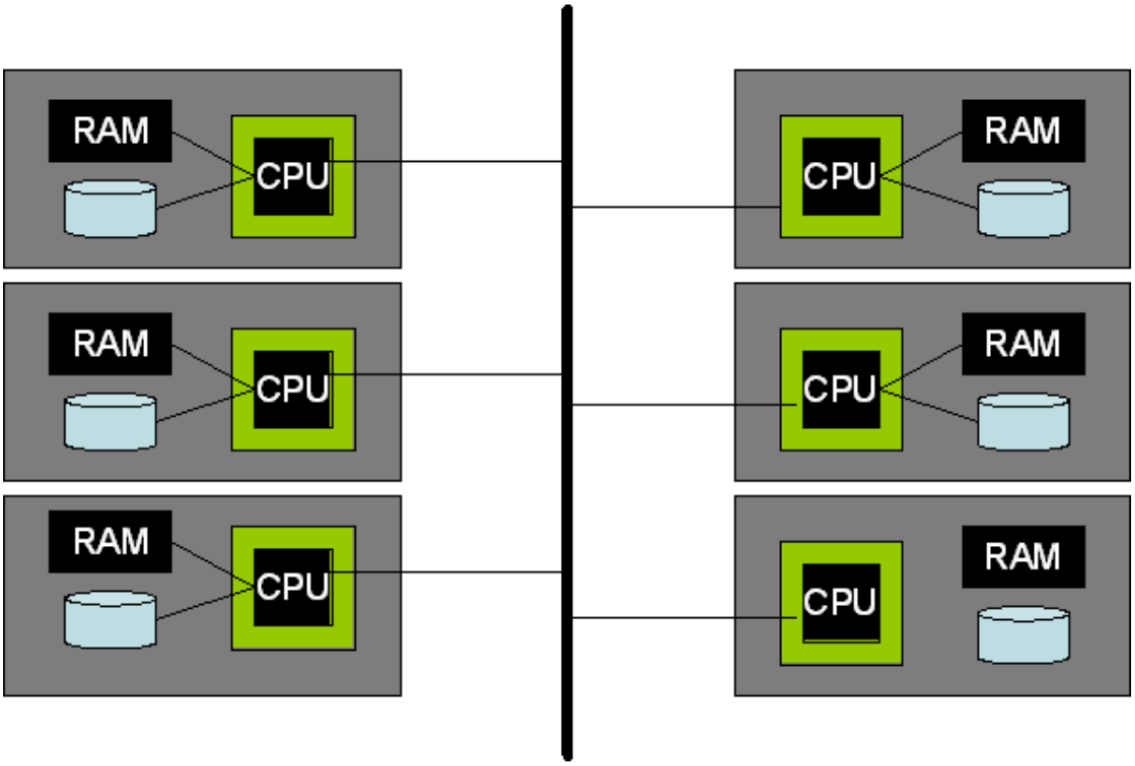
- Usually not the focus of typical papers
- Most considerations also applicable to any multi-user server system
- First, start with uniprocessor setup
- OS thread/process fundamentals:
    - OS process: each unit of execution has its private address space (memory)
    - OS thread: each unit of execution has shared address space, coordinated by OS.
        - Less overhead, less state, but less safety of memory
    - App level threads: same as OS threads, but coordinated by the app.
        - Even less overhead, but less safety (as in OS threads), and individual thread can hijack resources. DBMS can also implement threads
- Each DB client gets a "worker": one either has:
    - a process per DB worker
        - oldest approach, because most OS didn't have support for threads.
        - one issue is that there are common structures (buffer, lock table) so memory separation is actually counterproductive
        - still in use for some of the oldest DBs: IBM DB2, PostgreSQL, and Oracle (also pool version)
    - variant: process pool
        - variant of process per DB worker but reduces overhead
        - bounded size set of processes - we attach a free one to a client
        - also Oracle

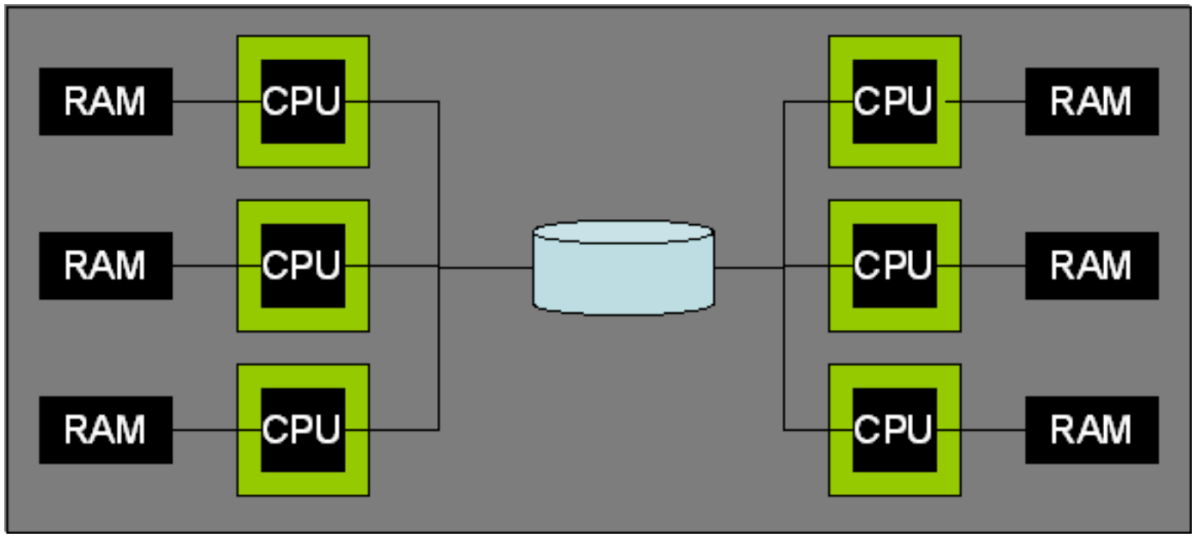
    - a thread per DB worker - within the DB process
        - "*OS does not protect threads from each other's memory overruns and stray pointers; debugging is tricky*"
        - could be implemented "app-level" or OS-level
        - used within Microsoft SQL Server (also pool version), MySQL (from survey)
        - most modern systems use some variant of this: Snowflake, Clickhouse, Databricks
    - variant: thread pool! also SQL Server, other modern systems
    - extreme version of thread per DB worker:
        - entire DB runs in a single process, thread per DB worker: DuckDB, SQlite (runs in all our phones!) - so no separate processes for each component.
- Shared resources - log, buffer pool, lock manager
    - in thread model, easy. In other models (typically) part of shared memory
- In general, in a modern multiprocessor system, we have n:m workers & requests:
    - where each worker can support multiple requests, and each request can be supported by multiple workers
    - for example, each client request can spawn multiple tasks across multiple workers (each a thread) - executed across multiple processors

Parallel Configs

Figures


CPU
CPU
CPU
RAM
CPU
CPU
CPU


RAM
CPU
RAM
CPU
RAM
CPU
CPU
RAM
CPU
RAM
CPU
RAM



Architectures

- Shared memory - aka a beefy machine
  - Many CPUs, shared RAM and disk
  - All three models run well in this setup - OS manages assignment of processes to processors
  - Variant with many CPUs: NUMA - different access costs for "close by" memory vs. "far away" memory.
  - Also even without NUMA, there is a hierarchy of caches L1 (fast, small), L2, L3 (slow, larger)
- Shared nothing
  - Many CPUs each with RAM and disk
    - each disk contains a portion of data typically
    - horizontal partitioning (round robin, hash, range)
  - Standard process model per CPU
  - Other issues: commit protocols (2PC), distributed deadlock detection, failures
- Shared disk
  - Many CPUs each with RAM and shared disk
  - Paper talks about a couple of systems
  - But this is actually a super popular design today:
    - Variant where each CPU has local disk, and shared disk is networked disk - shared object storage (GCS, S3)
    - Predominant design for CDWs: Snowflake, Databricks, BigQuery, Redshift...

Note: Got until here in class!

2) Query Processor

- Parsing/Authorization
  - Convert query into internal representation, make sure that all references are valid, and user is allowed to do the action
  - Consult catalog manager -
    - do the tables/attributes exist? are the operations valid on the attributes, given types?
  - Other non-obvious SQL correctness rules - Q: does anyone remember any?
    - every attribute in an aggregation query must be in a GROUP BY or aggregated
    - when we're doing set ops, schemas must be compatible
  - Authorization - is the user allowed read/write access? Can be pretty fine-grained

- Rewriting
  - expands views, simplify arithmetic expressions
  - add additional predicates: `R.x < 10 AND R.x = S.y -> S.y < 10` Q: why helpful?
  - join elimination - remove tables whose attributes are not in SELECT
    - `SELECT Emp.name, Emp.salary FROM Emp, Dept WHERE Emp.deptno = Dept.dno`
    - but must be careful wrt NULL values, multiplicities/multiset
  - flattening subqueries, aka query normalization
    - most optimizers operate on single SFW blocks
    - same issues: NULLs, multiplicities
  - Q: why all this necessary? Can't humans write better queries?
    - Turns out vast majority of queries are not written by humans, but by applications (BI tools, ORM modules), or by expanding views
- Optimization
  - essentially builds on System R query optimizer - next class!
  - SFW queries, other operators "afterwards" (grouping, ordering)
  - some "query plan" selected
  - various ways modern optimizers deviate from System R:
    - plan space
    - selectivity estimation
    - search algorithm
    - impl. in parallel hardware
  - will revisit all this in the next class
- Execution
  - Iterator model to implement all operators as subclasses (also known as the Volcano model)
    - `class iterator { ` iterator &inputs[]; void init();
      `tuple get_next();` void close();
      }````
  - Pull-based, tuple-at-a-time
  - later shown to not be optimal:
    - Overhead of tuple-at-a-time function calls
    - Limited ability to do hardware-level optimization (e.g., vectorization, hardware operator pipelining)
  - Early attempts to fix (eg MonetDB) materialized entire columns at a time - but giving away the benefits of database operator pipelining, also expensive in-memory
  - Ideal: `get_next` gives blocks of tuples (called morsel-driven parallelism) + compilation of query plans to support hardware-level pipelining across operators
- Where are the actual tuples?
  - Pointers to actual tuples in buffer pool pages, or copied over into a separate memory location
- Physical operators
  - access methods optionally include predicates as input, called a SARG (search argument following System R) - allowing for same form for both indexes, as well as seq scans (NULL)

Storage Configs

- DBs need spatial control (layout, e.g., clustered) and temporal control (when stuff is written out to disk, e.g., WAL) - had to do gymnastics with old OSs, but now much easier
  - Another issue of temporal control - double buffering - OS cache along with DB buffer - wasteful in memory, unnecessary copying of memory - can override via direct I/O, but important to know
- Buffer pool - set of frames of data in same format as disk - reduces ser/de cost.

    - Also a hash table with info on what's in the buffer pool - where it's from, whether it's dirty or pinned - and any page eviction stats
        - typically LRU but we seq scans do really badly with LRU (Q: why?) so need an exception
        - also special treatment of index pages (Q: why?)
- Typical setup:
    - file system with single large files per DB, use low-level OS interfaces to place and manipulate data
- Modern nuances:
    - modern "lakehouses" don't control how and where data is located, are simply asked to operate on it.
    - many write-heavy systems don't organize data: they just treat the log as storage (see LSM-trees), compact it over time

3) Storage Manager

- Multiple components:
    - Lock mgr
    - Log manager
    - Buffer pool
    - Access methods
- Essentially providing ACID guarantees:
    - durability through logging
    - atomicity, isolation through locking & logging (to abort txns)
    - consistency - runtime checks
- Serializability - implemented via one of three mechanisms:
    - Strict 2PL
    - MVCC - transactions don't hold locks; instead different transactions operate on various historical states of the database
    - OCC - all transactions update freely and maintain their read/write sets, aborted if they overlap with others