CMSC 724: Database Management Systems
Introduction/Background

Instructor: Amol Deshpande
amol@cs.umd.edu

Reading Papers

- Not a linear, cover-to-cover exercise
  - First goal: understand the **big picture**
  - Later goal: assess technical details and evidence
- You will not understand everything on first read
- Read strategically; revisit selectively

- Three-pass Method (Keshav)
  - **Pass 1 (5–10 min):** What is this paper about?
  - **Pass 2 (≈1 hour):** How does it work? Is the evidence convincing?
  - **Pass 3 (deep read):** Can I reconstruct, critique, or extend it?
  - Stop after an early pass when the paper is not relevant
  - Spend deep-reading time only where it pays off
    - Or if you are reviewing the paper, or for a class

Pass 1: Build the Map (5-10 mins)

- Read: title, abstract, introduction, conclusion
- Scan: section headings, figures, tables, references
- Identify the “five Cs”:
  - **Category:** system, algorithm, study, benchmark, vision?
  - **Context:** what prior work / problem?
  - **Contribution:** what is new?
  - **Correctness:** initially plausible?
  - **Clarity:** can you explain it simply?

Pass 2: Understand and Evaluate (1 hr)

- Read the design and evaluation carefully
- Start with the architecture diagram / running example
- For every graph: **What claim does it support?**
- Ask:
  - What assumptions does the system make?
  - What are the relevant baselines?
  - Are the comparisons fair?
  - What trade-off is being made?
  - When would this approach not work?

Pass 2: Understand and Evaluate (1 hr)

- Read the design and evaluation carefully
- Start with the architecture diagram / running example
- For every graph: **What claim does it support?**
- Ask:
  - What assumptions does the system make?
  - What are the relevant baselines?
  - Are the comparisons fair?
  - What trade-off is being made?
  - When would this approach not work?

Database-paper Checklist

- What database problem or workload motivates the work?
- Where does it fit in the stack?
  - Data model / language / optimizer / execution
  - Storage / indexing / transactions / distributed systems
- What is the central technical idea?
- What improves—and at what cost?
  - Latency, throughput, memory, I/O, freshness, generality
- What does the evaluation actually demonstrate?
- What is one limitation or next step?

Outline

- Overview
- Course Logistics
- AI
- Background: 424 Summary
- Architecture of a Traditional Database System
- Abstractions, Models, and Implementations
- Cross-cutting Issues in Data Management

- No laptop use allowed in the class !!

Outline

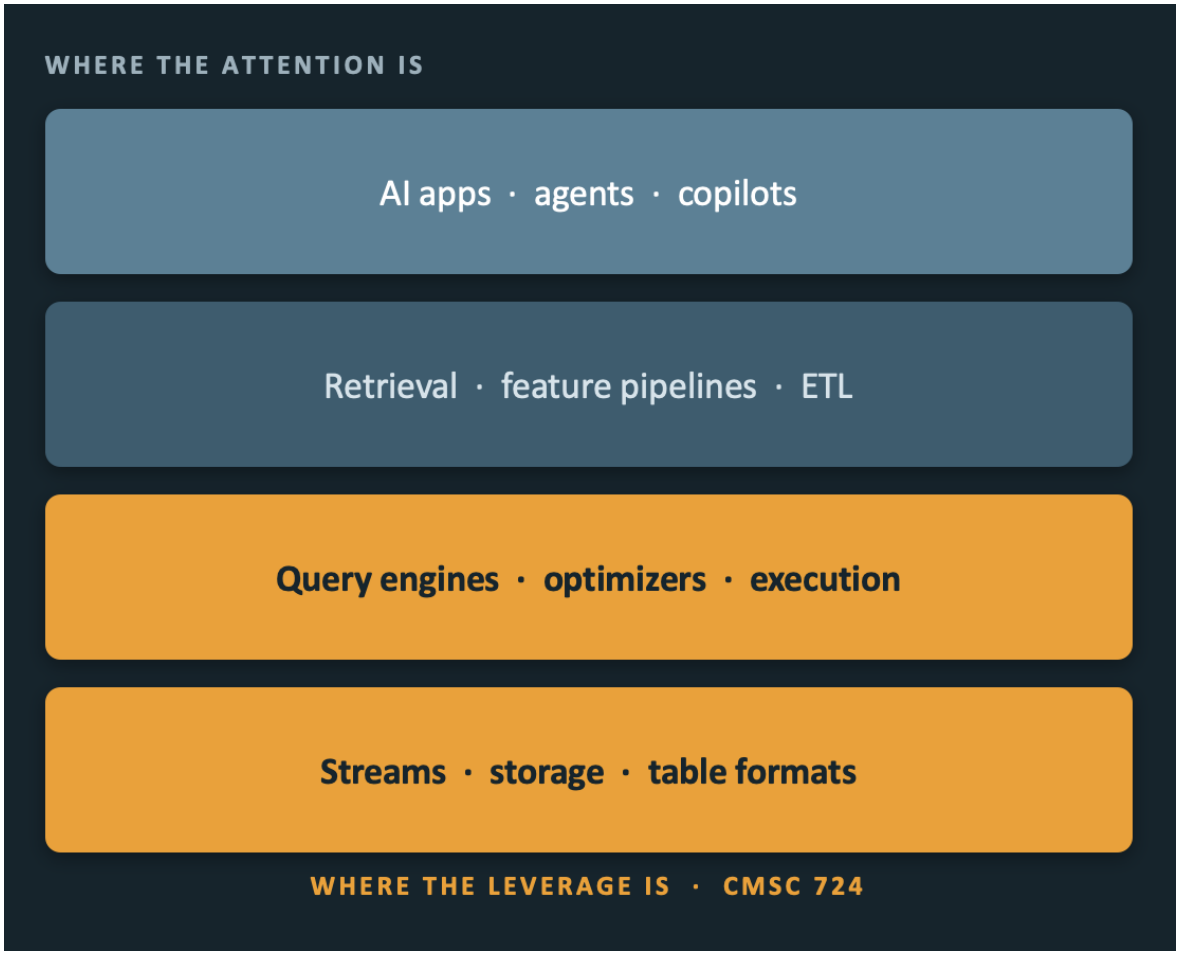
- Overview
- Course Logistics
- AI
- Background: 424 Summary
- Architecture of a Traditional Database System
- Abstractions, Models, and Implementations
- Cross-cutting Issues in Data Management

- No laptop use allowed in the class !!

Question for you

Given where AI tools will be two years from now: how do you think we should spend our time here? What would be the most valuable things for you to learn?

Why Take This Class?



Why Take This Class?

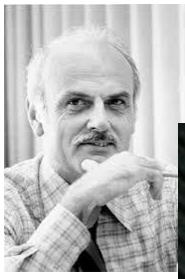
- Data infrastructure issues central to most AI infrastructure and applications

- Cannot supervise what you don't understand
  - Why is this system slow? Why did the cost triple? Does these benchmarks make sense?

- Tools have a half-life… ideas don't
  - Data management foundations will remain useful for a long time

Databases: A Brief History







60's

Shared "data banks" in military applications

Hierarchical and network (CODASYL) models

COBOL

IBM: IMS Hierarchical database for Apollo space program

IMS still in use today

70's

Ted Codd proposed the relational model (1969)

Two prototypes: INGRES (Stonebraker at Berkeley), and System R (IBM)

1976: Entity-Relationship Model (Peter Chen)

1977 Startup: Software Development Laboratories

Fierce debates between relational and CODASYL camps

80's

IBM DB2 (1983)

SQL became the standard query language (1986)

Many proposals for richer semantic models

Object-oriented, object-relational databases

Post(-in)gres Project at Berkeley (1986)

90's

World Wide Web (1989)

Parallel Databases

Data mining/OLAP

Lot of tooling for business analytics and app development

Client-server model, middleware

1996: PostgreSQL forked from Postgres codebase

XML

Data integration emerging as a key problem

Databases: A Brief History



00-05

Many companies in data warehousing/analytics (Aster Data, Greenplum, Vertica, Kickfire, …)

Columnar storage architectures (MonetDB, C-Store/Vertica)

Shared nothing (data center) systems

Main-memory based OLTP systems

06-10

Google MapReduce → Hadoop project at Yahoo

Key-value stores – simpler data model and no ACID, but much higher scalability

Many KV stores

Document databases (MongoDB, 2009)

Graph databases (Neo4j, 2007)

10's

Cloud-hosted SaaS offerings

Apache Spark (2011)

Both data warehouses (Snowflake, Redshift, etc.) and OLTP (Aurora, CosmosDB, CockroachDB)

Data Lakes becoming common (term coined 2011)

Fewer KV stores, but more other stores (graph, time-series, multi-model, vector)

20's

Blockchain DBs (not much uptake)

ML for databases (e.g., auto-tuning, NL → SQL)

Databases for ML

Unstructured analytics

LLMs for data integration, information extraction, etc.

Data Management Today

A large fraction of the data still in traditional DBMS systems

Still open and active research areas about improving performance, energy efficiency, new functionalities, changing hardware spectrum (SSDs) and so on…

Much of the data not stored in traditional database systems today

For a variety of fairly valid reasons

- Stream processing systems (focusing on *streaming* data)
- Special-purpose data warehousing systems (most start from some RDBMS)
- Batch analysis frameworks (like Hadoop, Pregel, Spark, …)
  - Typically data stored in distributed file systems
- Key-value stores (like HBase, Cassandra, Redis, …)
  - Basically persistent distributed hash tables
- Semi-structured/Document data stores (for XML/JSON query processing)
- Graph databases
- Scientific data management
- Machine learning data management
- Vector Databases

However, many lessons to be learned from database research

We see much reinvention of the wheel and similar mistakes being made as early on

What we will cover

A large fraction of the data still in traditional DBMS systems

- A deeper study of traditional RDBMS solutions (compared to 424)
- New functionalities/features
- Revisit some of the old design decisions (e.g., lay out data column-by-column instead of row-by-row, fully in-memory processing, etc)

Much of the data not stored in traditional database systems

- **Basic ideas behind, and why different from RDBMS:**
  - Stream processing systems
  - Special-purpose data warehousing systems
  - Batch analysis frameworks (specifically MapReduce)
  - Key-value stores (focus on the consistency issues)
- **If time permits:**
  - Semi-structured data stores
  - Graph databases

Databases: Thoughts

- Too many specialized data management systems at this time
- Leading to much silo-ed data stores that are can't really talk to each other well
- Building additional services (e.g., APIs) on top solves immediate problems, but adds more complexity over the long time
- Makes security, privacy, and governance issues much worse
- Need to figure out how to make things simpler
- Relational-like model + Schemas + Declarative Languages/Frameworks keep proving to be the winning combination
  - e.g., Apache Spark started as a map-reduce-like system, but SQL is the primary interface today

Course Overview

- We will cover:
  - A blend of classic papers + ongoing research (more focus on latter)
  - Reference book:
    - Readings in Database Systems, 5th edition. Mike Stonebraker, Joe Hellerstein, Peter Bailis.
  - Almost all papers are available online
  - Book contains some very nice overview chapters though – all available online at the book website (http://redbook.io)

- Prerequisite: CMSC 424
  - Class notes off my webpage

Course Structure

- Background + Overview (1 week)
- Data Models, Programming Abstractions (2 weeks)
- Storage Models (2 weeks)
- Query Processing + Optimization (5 weeks)
- Streaming Data Management + Dataflow Systems (2 weeks)

Learning Goals

- Intended to prepare you for data management research, broadly defined
  - Includes better understanding of data management issues in other fields
- Some specific goals:
  - You should be able to read, understand, and hopefully critique a data management paper
  - Given a new application domain, you should be able to:
    - ask the right questions to understand the key data management issues, and design/suggest appropriate solutions.
    - identify flaws (if any) with a proposed design or solution.
    - devise and reason about abstraction (independence) layers and their applicability to the application domain.
  - You should also have enough familiarity with how big data systems are built to be able to easily start using any of them, and reason about the observed performance of a deployed system, if only superficially.

Course Overview: Grading

- Background and setup assignments (5%)
  - Gradescope – you should already have access
- Programming assignments (2 * 5%)
  - Gradescope
- Paper presentations and discussions (3 * 15%)
  - Done as three “program committees”
  - Will use HotCRP – first set of assignments out soon
- Semester-long Research project (30%)
- Final (10%)

- See website for details

Deadlines at a glance

Due	What
Thu, Sept 4	Background questions + setup assignment Sept 11
Thu, Sept 24, 11:59pm	Paper critiques — session 1
Tue, Oct 6	Programming assignment: Query Optimization
Tue, Oct 13	Project: group members and proposal
Tue, Oct 20	Programming assignment: PostgreSQL internals
Tue, Oct 20	Project: anonymous peer reviews of proposals
Tue, Oct 27	Project: literature survey
Thu, Oct 29, 11:59pm	Paper critiques — session 2
Fri, Nov 20	Project: intermediate progress report
Tue, Nov 24, 11:59pm	Paper critiques — session 3
Mon, Dec 14	Project: final report

Course Overview: More Logistics

- Gradescope for assignments
- Slack for communication
  - Low volume – for project coordination, etc.
- See links on the webpage
- ChatGPT/Claude/Bard: Encouraged in most cases
  - Especially reading/understanding papers, or for all phases of class project
  - Don’t use when doing the assignments

Outline

- Motivation: Why study databases ?
- Course Logistics
- History of Databases
- Background: 424 Summary
- Architecture of a Traditional Database System
- Abstractions, Models, and Implementations
- Cross-cutting Issues in Data Management

- No laptop use allowed in the class !!

Why not use file systems to store data?

- Data redundancy and inconsistency
  - Multiple file formats, duplication of information in different files
- Difficulty in accessing data
  - Need to write a new program to carry out each new task
- Data isolation — multiple files and formats
- Integrity problems
  - Integrity constraints (e.g., account balance > 0) become “buried” in program code rather than being stated explicitly
  - Hard to add new constraints or change existing ones

Why not use file systems to store data?

- Atomicity of updates
  - Failures may leave database in an inconsistent state with partial updates carried out
  - Example: Transfer of funds from one account to another should either complete or not happen at all
- Concurrent access by multiple users
  - Concurrent access needed for performance
  - Uncontrolled concurrent accesses can lead to inconsistencies
    - Example: Two people reading a balance (say 100) and updating it by withdrawing money (say 50 each) at the same time
- Security problems
  - Hard to provide user access to some, but not all, data

DBMSs to the Rescue

- Provide a systematic way to answer many of these questions…
- Aim is to allow easy management of high volumes of data
  - Storing , Updating, Querying, Analyzing ….

- What is a Database ?
  - A large, integrated collection of (mostly *structured*) data
  - Typically models and captures information about a real-world ***enterprise***
    - Entities *(e.g. courses, students)*
    - Relationships *(e.g. **John** is taking **CMSC 424**)*
    - Usually also contains:
      - Knowledge of constraints on the data *(e.g. course capacities)*
      - Business logic *(e.g. pre-requisite rules)*
      - Encoded as part of the data model (preferable) or through external programs

DBMSs to the Rescue: Data Modeling

- Data modeling
  - **Data model**: A collection of concepts that describes how data is represented and accessed
  - **Schema:** A description of a specific collection of data, using a given data model

  - Some examples of data models that we will see
    - Relational, Entity-relationship model, XML…
    - Object-oriented, object-relational, semantic data model, RDF…

  - Why so many models ?
    - Tension between descriptive power and ease of use/efficiency
    - More powerful models → more data can be represented
    - More powerful models → harder to use, to query, and less efficient

DBMSs to the Rescue: Data Abstraction

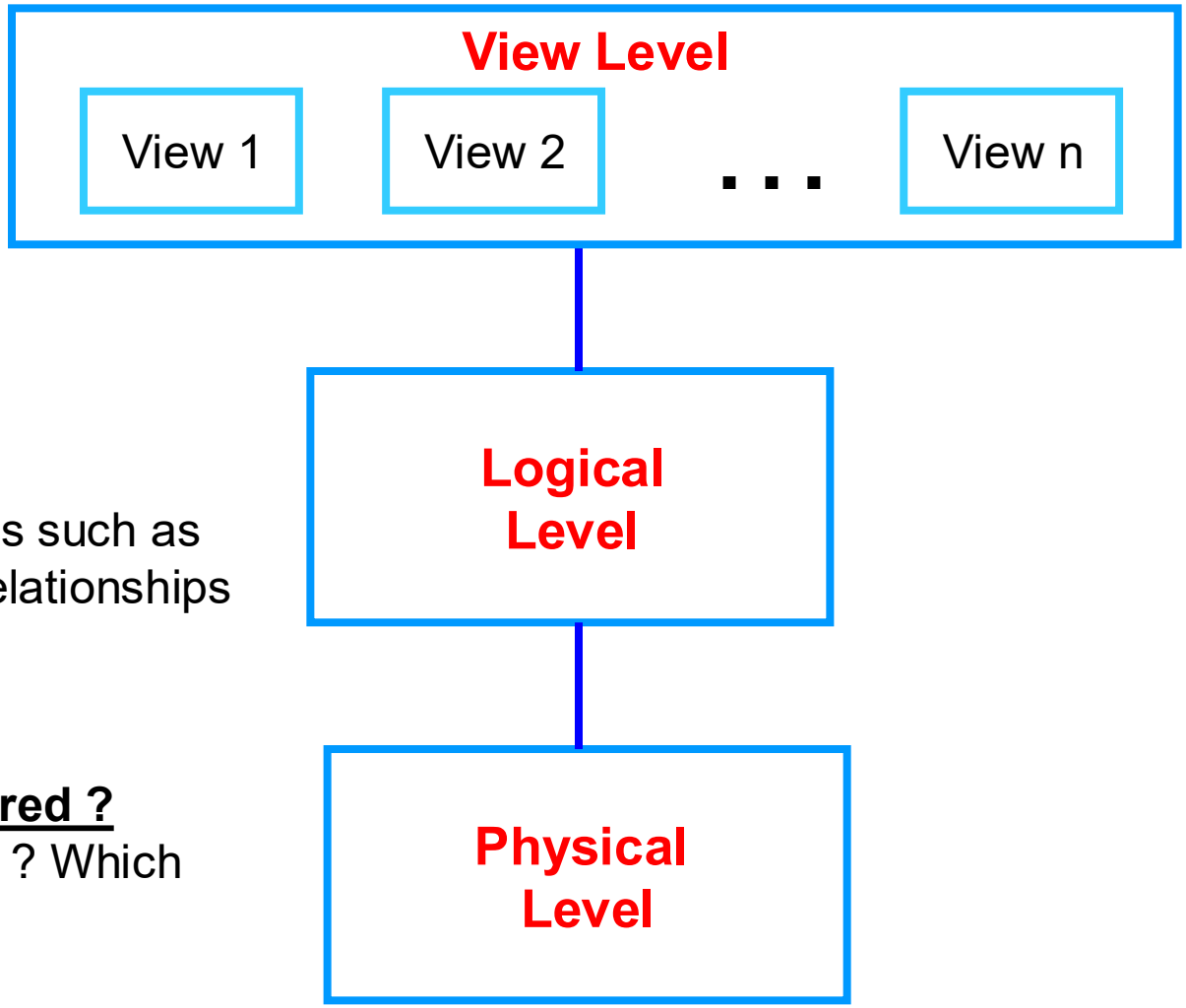
- Also called “Data Independence”

- Probably *the* most important purpose of a DBMS
- Goal: Hiding *low-level details* from the users of the system
  - Alternatively: the principle that
    - *applications and users should be insulated from how data is structured and stored*

- Through use of *logical abstractions*

Data Abstraction

What data users and application programs see ?



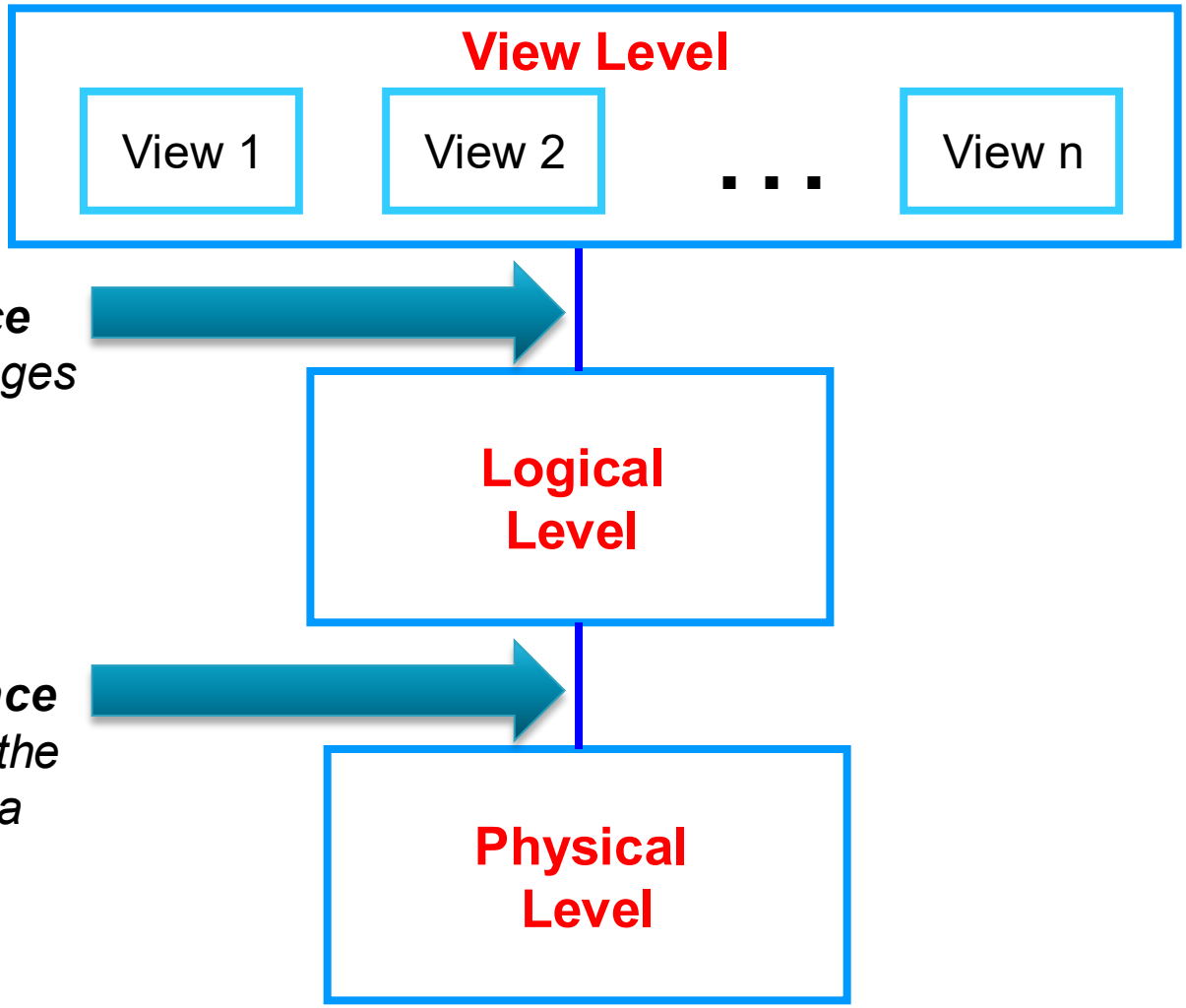
What data is stored ?

describe data properties such as data semantics, data relationships

How data is actually stored ?

e.g. are we using disks ? Which file system ?

Data Abstraction



Logical Data Independence
Protection from logical changes to the schema

Physical Data Independence
Protection from changes to the physical structure of the data

What about a Database System ?

- A DBMS is a software system designed to store, manage, facilitate access to databases

- Provides:
  - Data Definition Language (DDL)
    - For defining and modifying the schemas
  - Data Manipulation Language (DML)
    - For retrieving, modifying, analyzing the data itself
  - Guarantees about correctness in presence of failures and concurrency, data semantics etc.

- Common use patterns
  - Handling transactions (e.g. ATM Transactions, flight reservations)
  - Archival (storing historical data)
  - Analytics (e.g. identifying trends, **Data Mining**)

Basic topics covered in 424

- representing information
  - data modeling
  - semantic constraints
- languages and systems for querying data
  - complex queries & query semantics
  - over massive data sets
- concurrency control for data manipulation
  - ensuring transactional semantics
- reliable data storage
  - maintain data semantics even if you pull the plug
  - fault tolerance

Basic topics covered in 424

- representing information
  - data modeling: *relational models, E/R models*
  - semantic constraints: *integrity constraints, triggers*
- languages and systems for querying data
  - complex queries & query semantics*: SQL, MongoDB, Spark*
  - over massive data sets*: indexes, query processing, optimization*
- concurrency control for data manipulation
  - ensuring transactional semantics: *ACID properties*
- reliable data storage
  - maintain data semantics even if you pull the plug: *durability*
  - fault tolerance: *RAID*

Relational Data Model

- Most widely used model today
- Main concepts:
  - *relation*: basically a table with rows and columns
  - *schema* (of the relation): description of the columns
- Example:

  courses(dept char(4), courseID integer, name varchar(80), instructor varchar(80))

  students(sid char(9), name varchar(80), …)

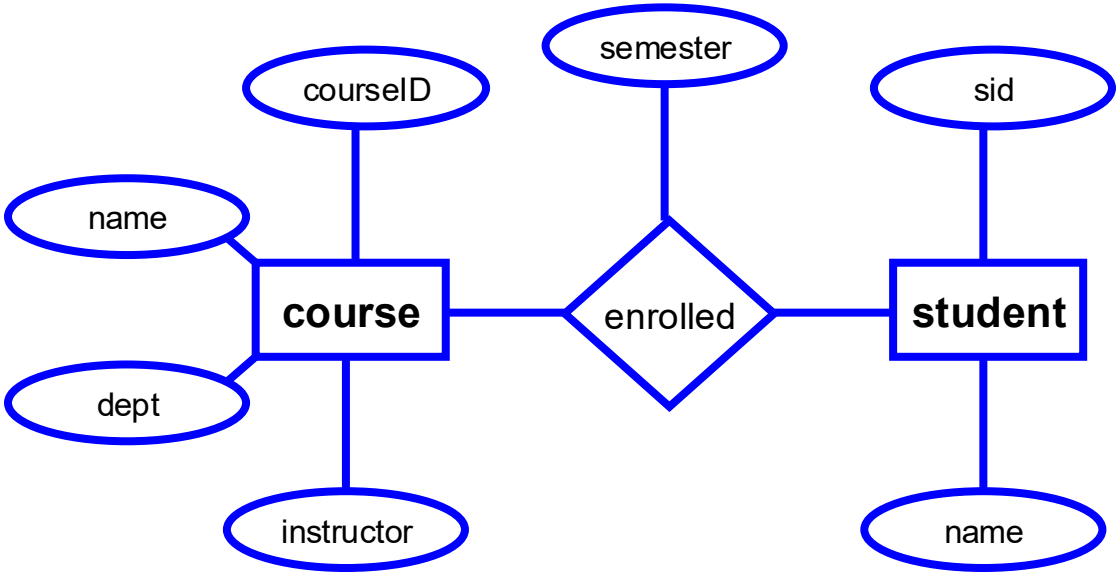
  enrolled(sid char(9), courseID integer, …)
- This is pretty much the only construct

An *instance* of the courses relation

Dept	CourseID	Name	Instructor
CMSC	424	…	…
CMSC	427	…	…

Entity-Relationship (E/R) Data Model

- More powerful model, commonly used during conceptual design
  - Easier and more intuitive for users to work with in the beginning
- Has two main constructs:
  - Entities: e.g. courses, students
  - Relationships: e.g. enrolled
- Diagrammatic representation



Relational Query Languages

- Example schema: *R(A, B)*
- Practical languages
  - SQL
    - select A from R where B = 5;
  - Datalog (sort of practical) – Has seen a resurgence in recent years
    - q(A) :- R(A, 5)
- Formal languages
  - Relational algebra

    $\pi_A (\sigma_{B=5} (R))$ -- You will encounter this in many papers
  - Tuple relational calculus

    $\{ t : \{A\} \mid \exists s : \{A, B\} (R(A, B) \wedge s.B = 5) \}$
  - Domain relational calculus
    - Similar to tuple relational calculus

Relational Query Languages

- Important thing to keep in mind:
  - SQL is not SET semantics, it is BAG semantics
  - i.e., duplicates are not eliminated by default
    - With the exception of UNION, INTERSECTION, MINUS
  - Relational model is SET semantics
    - Duplicates cannot exist by definition
- Relational algebra: Six basic operators
  - Select (σ), Project (π), Carterisan Product (×)
  - Set union (U), Set difference (-)
  - Rename (ρ)

Join Variations (SQL and Relational Alg.)

- Tables: r(A, B), s(B, C)

name	Symbol	SQL Equivalent	RA expression
cross product	$\times$	select * from r, s;	$r \times s$
natural join	$\bowtie$	natural join	$\pi_{r.A,\ r.B,\ s.C}\sigma_{r.B\ =\ s.B}(r\ x\ s)$
theta join	$\bowtie_\theta$	from .. where θ;	$\sigma_\theta(r\ x\ s)$
equi-join	$\bowtie_\theta$ *(theta must be equality)*		
left outer join	$r \ ⟕ \ s$	left outer join (with "on")	(see previous slide)
full outer join	$r \ ⟗ \ s$	full outer join (with "on")	–
(left) semijoin	$r \ltimes s$	none	$\pi_{r.A,\ r.B}(r \bowtie s)$
(left) antijoin	$r \triangleright s$	none	$r - \pi_{r.A,\ r.B}(r \bowtie s)$

Relational Model: Normalization

- Goal: What is a “good” schema for a database? How to define and achieve that

- Problems to avoid:
  - Repetition of information
    - For example, a table:
      - *accounts(owner_SSN, account_no, owner_name, owner_address, balance)*
    - Inherently repeats information if a customer is allowed to have more than one account
  - Avoid set-valued attributes

Relational Model: Normalization

1. Encode and list all our knowledge about the schema
   - Functional dependencies (FDs)

     SSN → name (means: SSN "implies" name)
   - If two tuples have the same "SSN", they must have the same "name"

     movietitle → length ???? Not true.
   - But, (movietitle, movieYear) → length --- True.
2. Define a set of rules that the schema must follow to be considered good
   - "Normal forms": 1NF, 2NF, 3NF, BCNF, 4NF, …
   - A normal form specifies constraints on the schemas and FDs
3. If not in a "normal form", we modify the schema

See 424 class notes for more

Semantic Constraints

- SQL supports defining integrity constraints over the data
  - Basically a property that must always be valid
  - E.g., a customer must have an SSN, a customer with a loan must have a sufficiently high balance in checking account, etc.

- Triggers
  - If something happens, then execute something
    - E.g., if a tuple inserted in table *R,* then update table *S* as well
  - Quite frequently used in practice, and surprising not as well optimized for large numbers

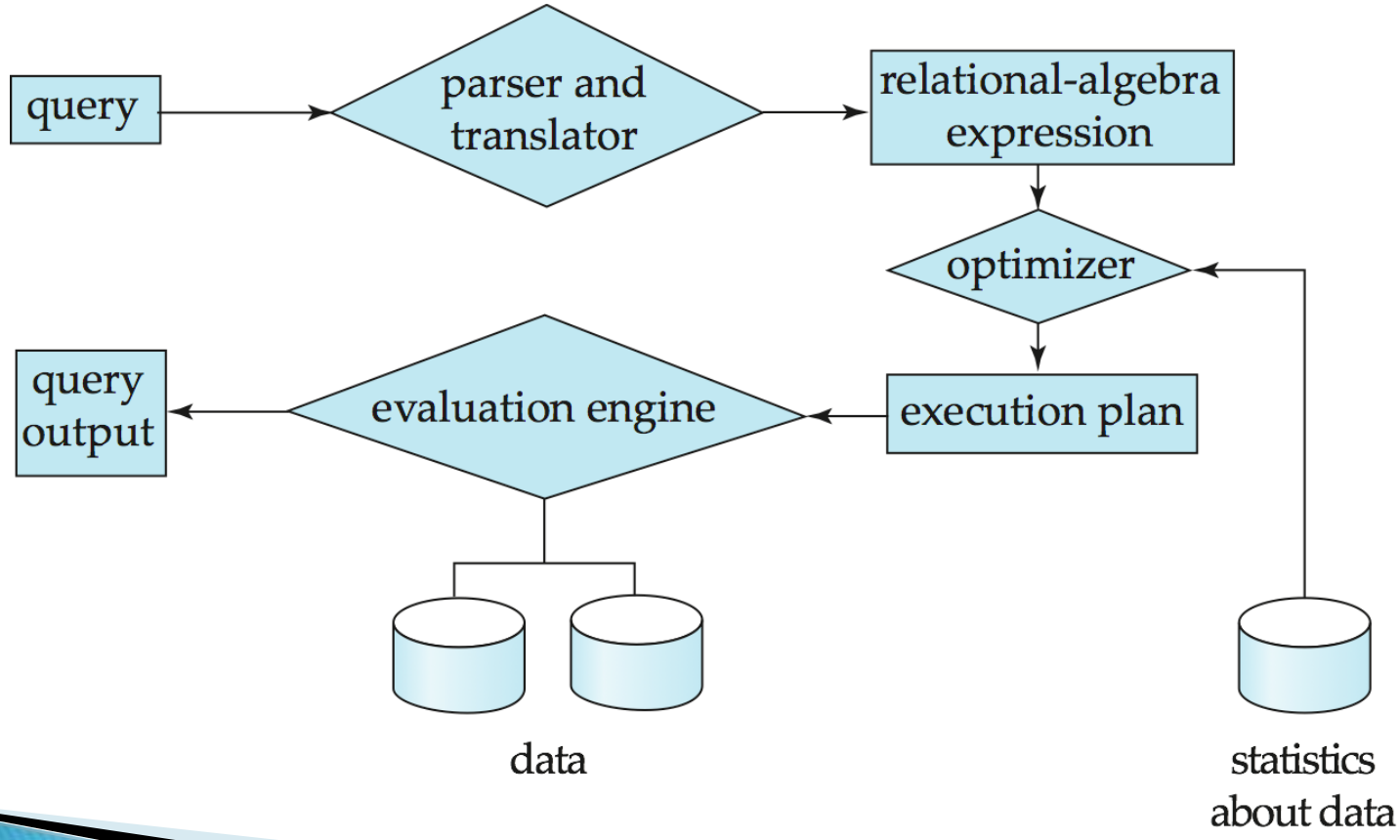
Storage

- Storage:
  - Need to be cognizant of the memory hierarchy
    - Many of traditional DBMS decisions are based on:
      - Disks are cheap, memory is expensive
      - Disks much faster to access sequentially than randomly
    - Much work in recent years on revisiting the design decisions…
  - RAID: Surviving failures through redundancy

- Indexes
  - One of the biggest keys to efficiency, and heavily used
  - **B+-trees** most popular and pretty much the only ones used in most systems
  - Others: R-trees, kD-trees, …

Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



Parallel and NoSQL

- Parallel and Distributed Environments
  - Shared-nothing vs Shared-memory vs Shared-disk
  - Speedup vs Scaleup
- How to ”parallelize” different relational operations
- Motivation for emergence of NoSQL Systems
- Map-reduce Framework for Large-scale Data Analysis
- Apache Spark: Resilient Distributed Dataset (RDD) Abstraction
- MongoDB
  - JSON Data Model
  - MongoDB Query Language

Transactions

- *Transaction*: A sequence of database actions enclosed within special tags
- Properties:
  - ***Atomicity***: Entire transaction or nothing
  - ***Consistency***: Transaction, executed completely, takes database from one consistent state to another
  - ***Isolation***: Concurrent transactions *appear* to run in isolation
  - ***Durability***: Effects of committed transactions are not lost
- Consistency: programmer needs to guarantee that
  - DBMS can do a few things, e.g., enforce constraints on the data
- Rest: DBMS guarantees

Transactions: How?

- ***Atomicity***: Through "logging" of all operations to "stable storage", and reversing if the transaction did not finish
- ***Isolation***:
  - Locking-based mechanisms
  - Multi-version concurrency control
- ***Durability***: Through "logging" of all operations to "stable storage", and repeating if needed

- Some key concepts:
  - Serializability, Recoverability, Snapshot Isolation, Two-phase locking, Write-ahead logging, …

Next class…

- We will cover some of the key topics from the "Architecture" paper, and discuss some of the broader data management issues

- First 3 programming assignments will be posted right away (will be due over the next 4-6 weeks)
  - Generally we are quite flexible about these assignments

- First written assignment will be out soon as well
  - Focusing on first 2-3 readings

Outline

- Motivation: Why study databases ?
- Course Logistics
- History of Databases
- Background: 424 Summary
- Abstractions, Models, and Implementations
- Architecture of a Traditional Database System

- No laptop use allowed in the class !!

What is a graph database?

A graph database is a database designed to store data as **nodes and edges**, rather than primarily as tables and rows. It's especially useful when relationships are central to the application—like social networks, fraud detection, or recommendation systems—because traversing those relationships is much more natural and efficient than doing

What is a document database?

A document database stores data as self-contained documents, usually in a JSON-like format, rather than splitting it across normalized tables. It's useful when records have flexible or nested structures, and it often lets developers store application objects in a form closer to how they appear in code.

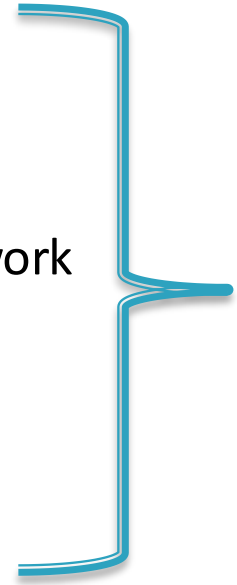
What is a distributed database?

A distributed database is a database where the data is stored across multiple machines rather than on a single server. The machines work together so the system appears to applications as one database, often to improve scalability, availability, or performance.

Common conflation	What gets mixed together
Relational Database = SQL Database	Data model ↔ query language
Relational Query Language = SELECT/FROM/WHERE syntax	Language semantics ↔ surface syntax
Relationships = joins	Logical relationship ↔ operator used to recover it
Schemaless = there is no schema	Who/what enforces structure ↔ whether structure exists
Vector database = database that uses ANN indexes	Logical capability ↔ specialized index/algorithm

Design Dimensions for a DMS

- User-facing
  - Data Model
  - Query Language and/or Programming Framework
  - Transactions
  - Performance Guarantees/Focus
  - Consistency Guarantees
- Implementation
  - In-memory and at-rest storage representations
  - Target Computational Environment
  - Query processing and optimization
  - Transactions' implementation
  - Support for streaming, versioning, approximations, etc.



These "define" the "type" of the database

Data Models

- A collection of concepts that describes how data is represented and accessed
  - **Schema:** A description of a specific collection of data, using a given data model
- Goal is to capture the properties of the data at the "right level"
  - Too strict → may not be able to store the data we want
  - Too loose → may not be able to build a query language on top, or efficiently optimize
- Examples:
  - Relational, Entity-relationship model, XML, JSON…
  - Object-oriented, object-relational, semantic data model, RDF…
  - Sets of "objects", ML models

Query Languages/Frameworks

- Define how to go from input data, to some desired output
  - Depends to some extent on the data model, but still a lot of flexibility
- Want this to be as ”high-level” or “declarative” as possible
  - Too high-level → fewer use cases will be covered
  - Too low-level → harder to use, support or optimize
  - Lot of work on trying to find the “right” level of abstraction
  - Interest in formally defining the power of a language, etc.
- Examples:
  - SQL: Input relations → output relations
  - Apache Spark RDD or Map-Reduce: Input “set of objects” → output ”set of objects”
  - BlinkDB: Input relations + approximation guarantees → output relations
  - Visualization Tools: Input datasets → Plots
- If supporting “streaming” or “versioning” or “approximations”, need to define what that means

Transactions/Updates (User-facing)

- Support for updating the data in the DMS
  - Some of the same issues as query language w.r.t. the expressiveness of the language
- Some considerations:
  - Consistency guarantees around updates (ACID or not)
    - Becomes more complicated in the distributed setting, with replication and sharding/partitioning
  - Batch updates vs one-at-a-time (impact on staleness)
  - Immutability: guarantees around no-tampering (e.g., blockchains)
  - Versioning: ability to support multiple branches, and ”time-travel”
- If the language is not expressive enough, have to do more work in the applications → impact on guarantees
  - e.g., MongoDB (and many other NoSQL stores) didn’t support multi-collection updates for a long time

In-memory and at-rest storage representations

- How is data laid out on disks (at rest) and in-memory, and across machines
  - Significant impact on performance
  - Depends somewhat on data model, but not fully ("Data Independence")
  - May use different representations when loading in memory (serialization/deserialization cost)
  - Usually we also build "indexes" for efficient search
  - Transmission over network also a concern

- Some options:
  - Row-oriented storage for relational model
    - Traditional approach: good for updates but bad for queries
  - Column-oriented storage for relational model
    - Really good performance for queries, but updates not easy to handle
  - Object storage (e.g., with pointers) for object-oriented databases or Graph databases
    - Pointers don't translate from disk to memory easily
  - Hierarchical storage for JSON/XML
  - Structured file formats like CSV (row), Parquet (columnar) for Data Lakes
    - Less up-front cost of "ingesting" the data, but more complex and less efficient to support
    - Harder to put any "structure" or "data model" on top of it



"Data Independence" → not "required" to, e.g., use pointers for graph databases – easy to convert to row-oriented storage

- Thoughts:
  - Cost of "ingest" must be amortized over many uses – for one-time use of data, prefer to leave in its native format

Target Computational Environment

- Many, many combinations here
  - Single machine vs parallel (locally) vs geographically distributed
  - Hardware
    - e.g., multi-core vs many-core, large-memory, disks or SSDs, RDMA, cache assumptions, and so on
  - Use of cloud/virtualization
    - Can have a significant impact on performance guarantees
    - Also, may put limits on what can be done (e.g., if using “serverless functions”)
- Hard to build a different system for each combination
- Increasing interest in “auto-tuning” through use of ML
  - Try to ”learn” how to do things for a new environment

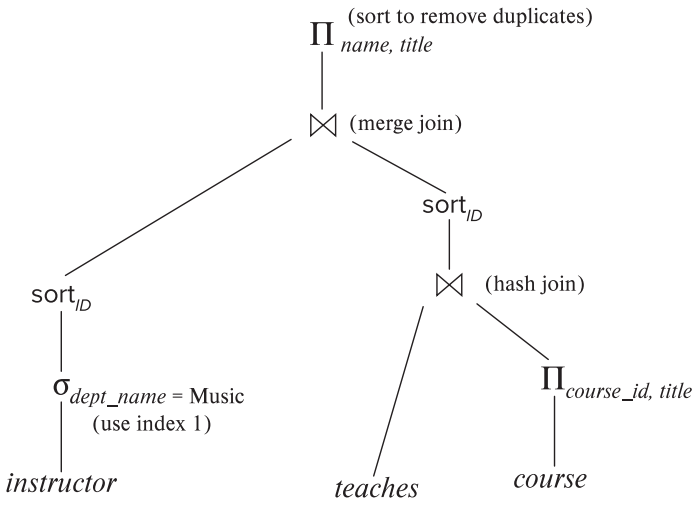
Query Processing and Optimization

- Depends significantly on how “declarative” is the query language/framework
- Most systems support a collection of low-level “operators”
  - Relational: joins, aggregates, etc.
  - Apache Spark: map, reduce, joins, group-by, …
- Should choose a good set of operators
  - Restricts the optimization abilities
  - e.g., if only support “binary” joins then lose the ability to optimize multi-way joins
  - In general, a sequence of operations will perform worse than a single equivalent operation
- Need to map from the overall “task” or “query” into those low-level operators
  - Usually called a “query execution/evaluation plan”
  - There may potentially be many many ways to do this (depending on how declarative)
  - Try to choose in a “cost-based” manner
    - Need the ability to estimate costs of different plans
    - ”Heuristics” often preferred in less mature systems

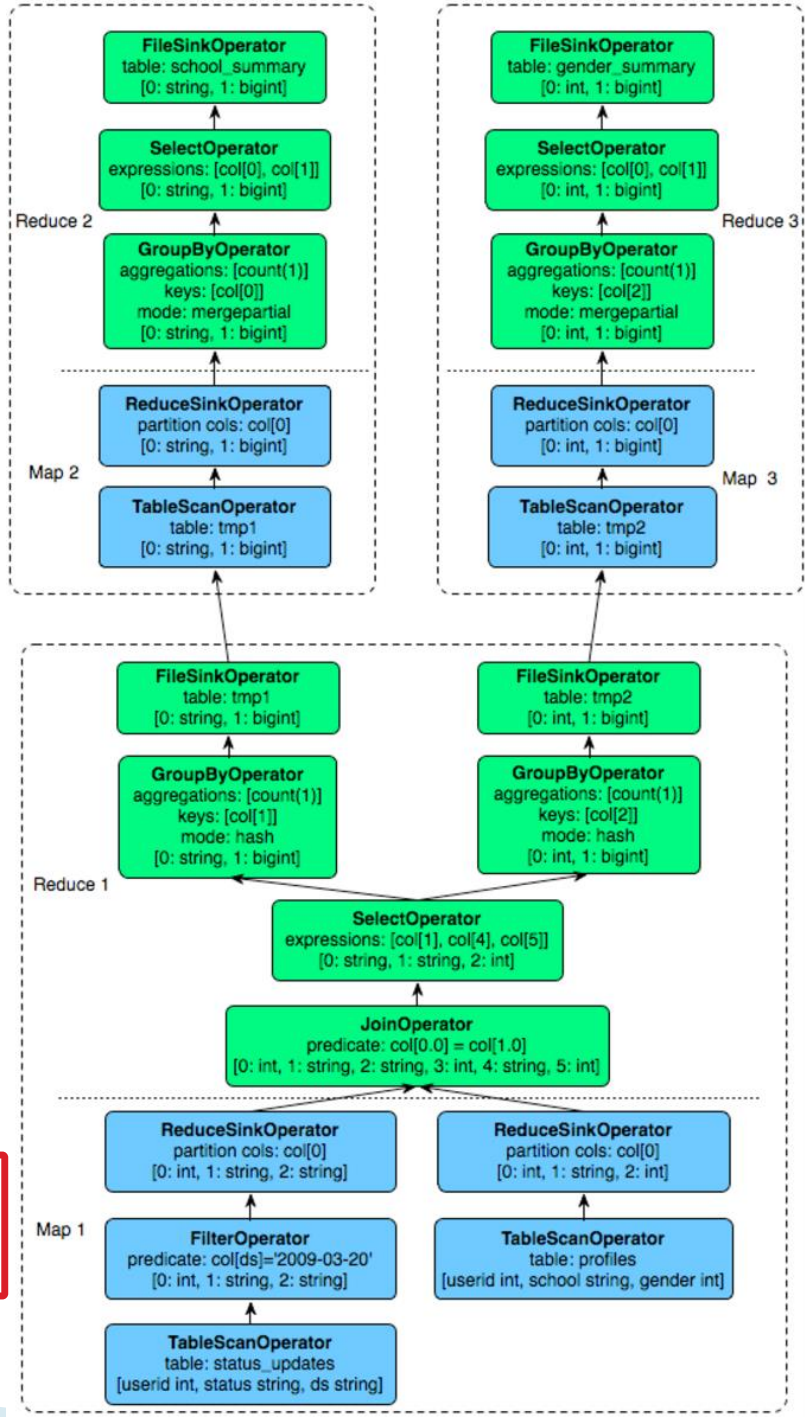
Query Processing and Optimization

- Cost measure
  - Important to decide what resource you are optimizing
  - Need to focus on the bottlenecks of the environment
  - Traditionally: CPU, Memory, Disks
  - Today, network costs play a very important role
  - Also: optimizing for “total resources” or “wall-clock time” ?
    - Especially important in parallel/distributed environments
- May wish to “pre-compute” certain queries to reduce the query execution times
  - Especially for ”real-time” queries over “streaming” data
  - Often called “materialized views” in the context of relational databases
  - Any pre-computed data must be kept up-to-date
- Adaptive query processing
  - May wish to “change” the query plan during execution based on what we are seeing

Query Plans vs...

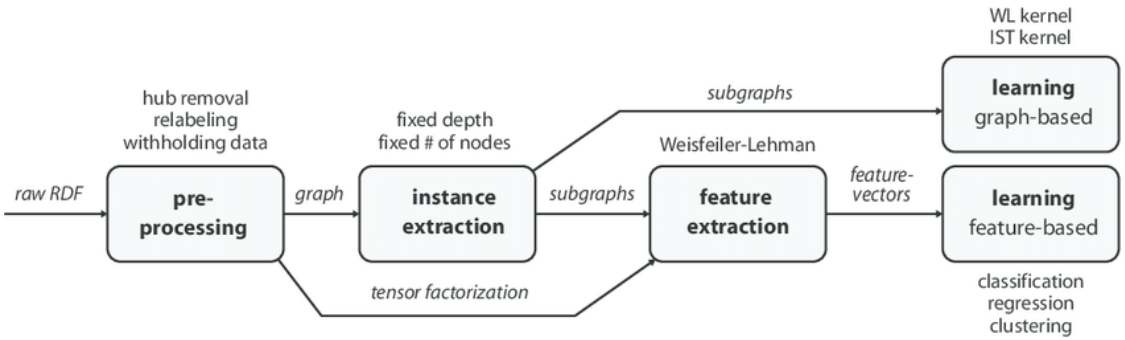


SQL "Query Plan"



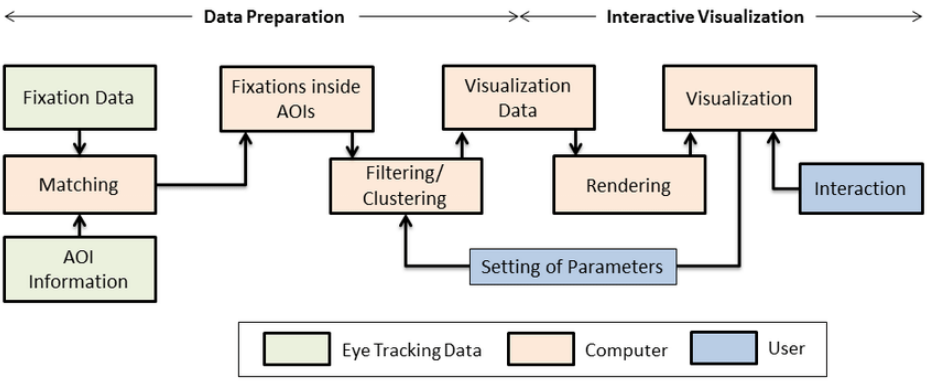
Apache Hive "Query Plan"
(Hive is an SQL layer on top of Hadoop)

vs … Data Transformation Pipelines

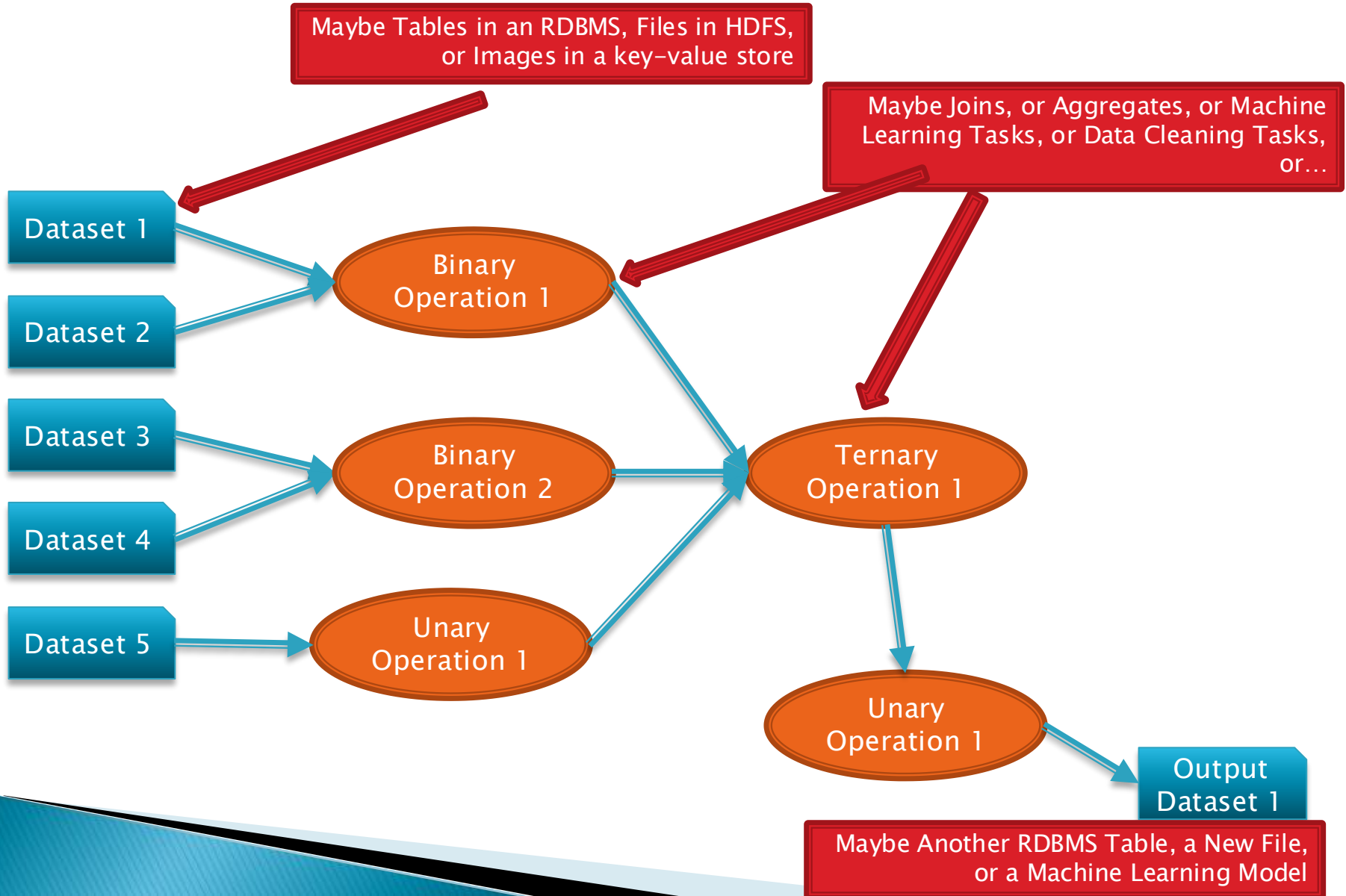


Machine Learning Pipeline

Data Preparation and Visualization Pipeline



Many similarities across systems…



Support for Streaming, Versioning, Approximations, etc…

- Streaming
  - Usually need to keep a lot of pre-built state to handle high-rate data streams
  - Each new update → modify the pre-built state, and output results
  - Hard to do this in a generic way
    - A specialized system will likely have much lower response times (e.g., in financial settings)
- Versioning
  - So far, the focus has primarily been on storage (i.e., how to compactly store the version history over time)
  - The "retrieval" of old versions considered less important to date
- Immutability
  - More interest in recent years on this, but still pretty open from a database perspective
- Approximate Query Processing
  - Usually need additional constructs like "random samples"

Recap

- Not intended to cover all data management research, but as a helpful guide to think about data management systems
  - Data cleaning, visualizations, security, privacy, …
- Finding the right abstractions is often the key to wide usage
- More complex abstractions may provide short-term wins, but often become difficult to manage and use over time
- Implementations have become very complex and involved today
  - Easy to obtain significant benefits focusing on a specific workload and hardware
  - But hard to get, and/or reason about performance in general settings
  - Experimental evaluations can’t cover all different scenarios

Outline

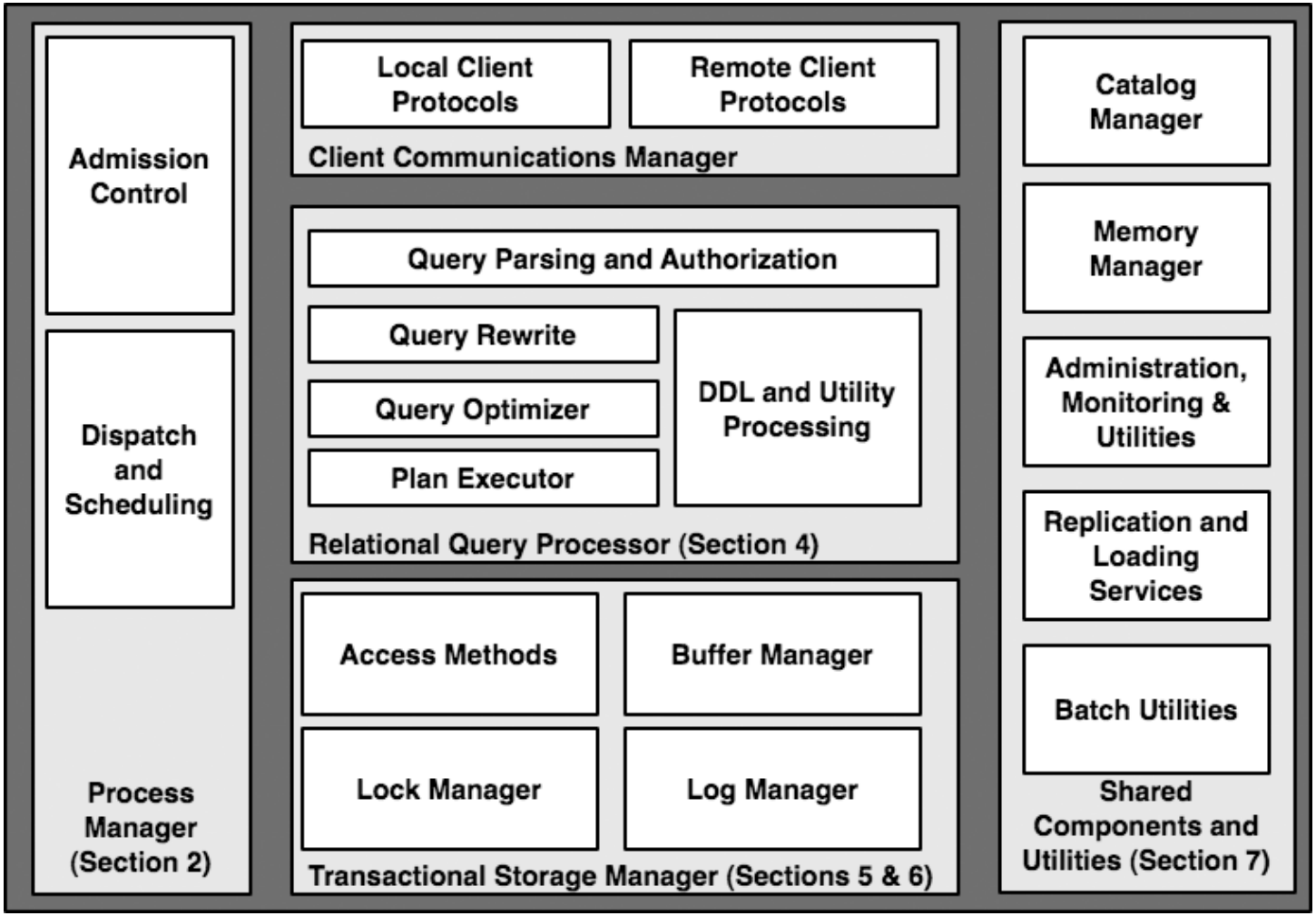
- Motivation: Why study databases ?
- Course Logistics
- History of Databases
- Background: 424 Summary
- Abstractions, Models, and Implementations
- Architecture of a Traditional Database System

- No laptop use allowed in the class !!

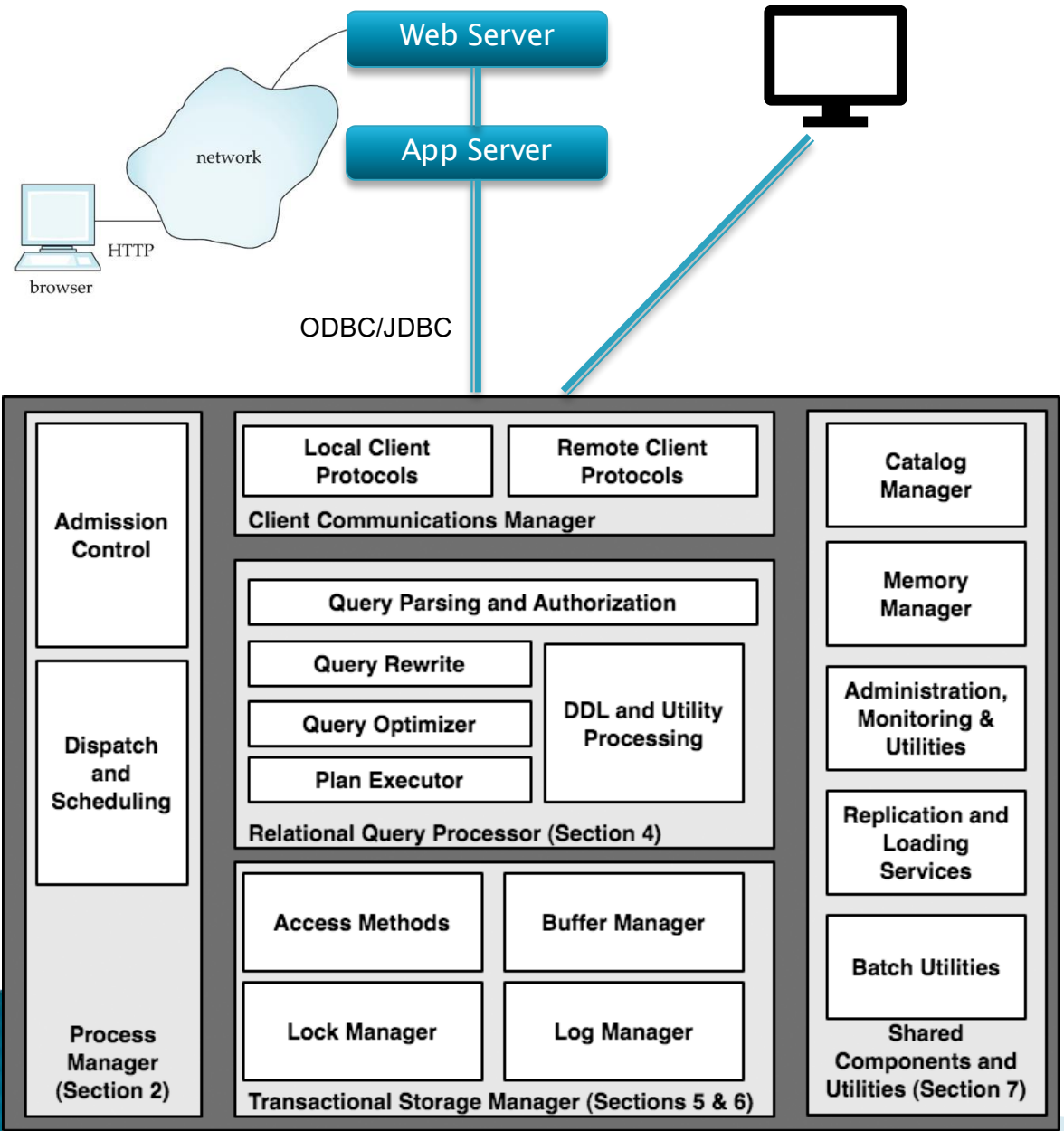
Architecture of a Traditional DBMS

- Paper by: Hellerstein, Stonebraker, Hamilton
- Covers the main components of a typical relational DBMS
- Goals for today:
  - Discuss an end-to-end system and issues like admission control, process models, etc.
  - Won't go deep into query processing, transactions, etc. – that will be later

Main Components



Life of a Query



Clients connect using standard or proprietary protocols to submit "queries"/"transactions"

Admission Control

Assign a "thread of computation"

Parse, compile, optimize the query

Start fetching or updating the data
- get locks
- create log records if needed
- etc…

Return data batch-at-a-time

Process Models

- Question: How do we handle multiple user requests/queries “concurrently”?

- Lot of variations across Operating Systems
  - OS Process: Private address space – scheduled by kernel
  - OS (Kernel) Thread: Multiple threads per process – shared memory
    - Support for this relatively recent (late 90’s, early 00’s)
    - OS can “see” these threads and does the scheduling
  - Lightweight threads in user space
    - Scheduled by the application
    - Need to be very very careful, because OS can’t pre-empt
    - e.g., can’t do Synchronous I/O
  - DBMS Threads
    - Similar to general lightweight threads, but special-purpose

Process per DBMS Worker

- Each query gets its own “process” (e.g., PostgreSQL, IBM D2, Oracle)*
  - Heavy-weight, but easy to port to other systems
  - Need support for “shared memory” (for lock tables, etc)

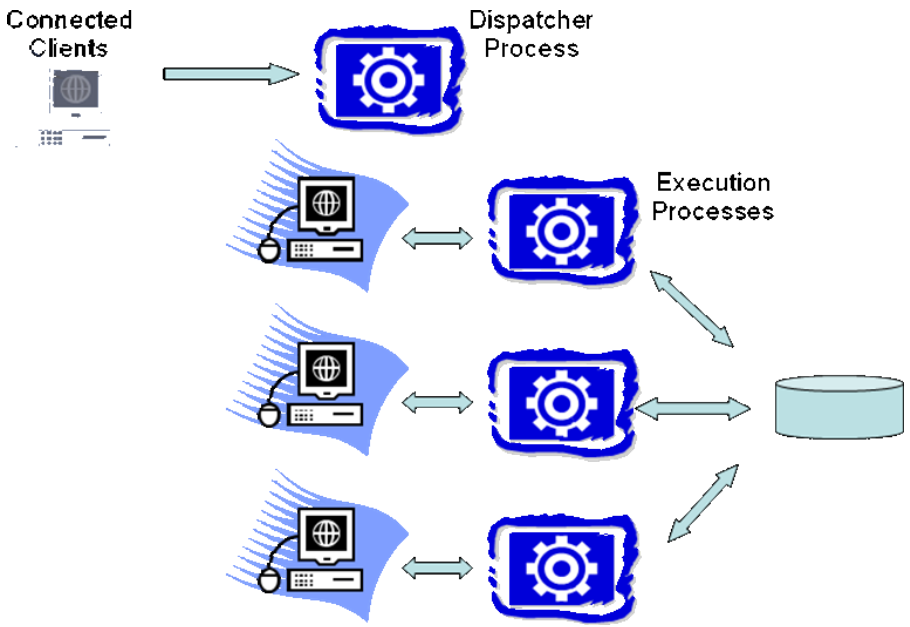


Fig. 2.1 Process per DBMS worker model: each DBMS worker is implemented as an OS process.

* All circa 2007 – may have changed since then.

Thread per DBMS Worker

- A single-multithreaded server
  - Need support for “asynchronous” I/O (so threads don’t block)
  - Easy to share state, but also makes it easy for queries to interfere

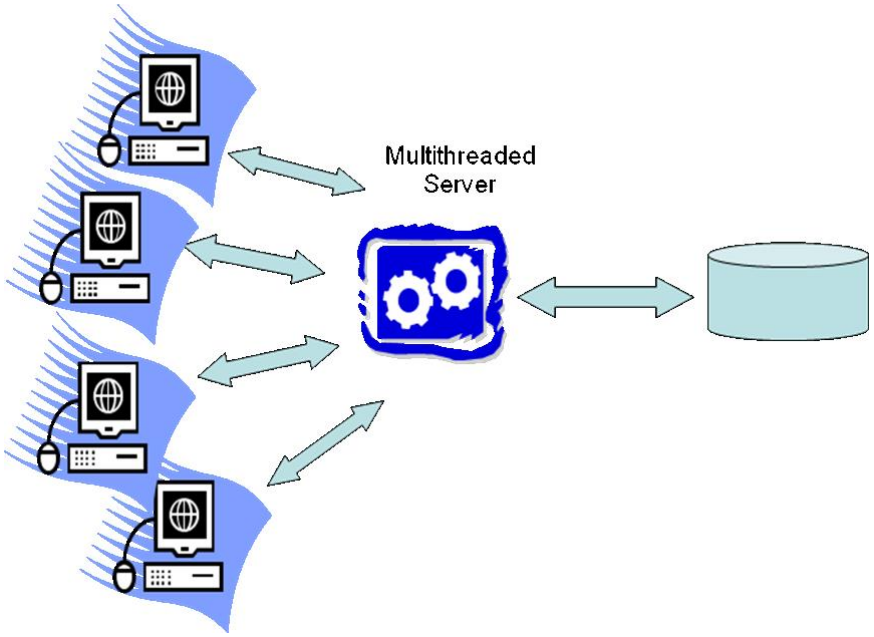


Fig. 2.2 Thread per DBMS worker model: each DBMS worker is implemented as an OS thread.

Process (or Thread) Pools

- Typically DBMS allots a pool of processes or threads, and multiplexes clients/requests across those

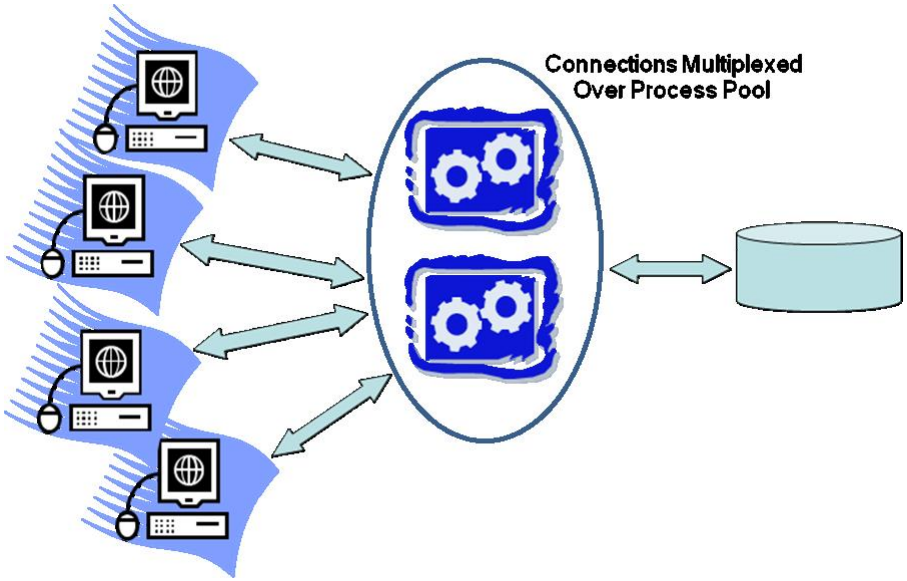


Fig. 2.3 Process Pool: each DBMS Worker is allocated to one of a pool of OS processes as work requests arrive from the Client and the process is returned to the pool once the request is processed.

Shared Data Structures

- Buffer Pool
  - Manages the disk blocks that are currently being used by the different workers
  - Use some replacement strategy like Least-recently-used
- Log Tail
  - All updates generate “log” records that need to properly numbered and flushed to disk
- Lock Table
  - For synchronization across workers in case of conflicts
- Client Communication Buffers
  - To keep track of what data has already been sent back to clients, and to buffer more outputs

Parallel Architectures

- Shared-memory and shared-nothing architectures prevalent today
- Shared-memory: easy to evolve to because of shared data structures
- Shared-nothing: require more coordination
  - Data must be partitioned across disks, and query processing needs to be aware of that
  - Single-machine failures need to be handled gracefully

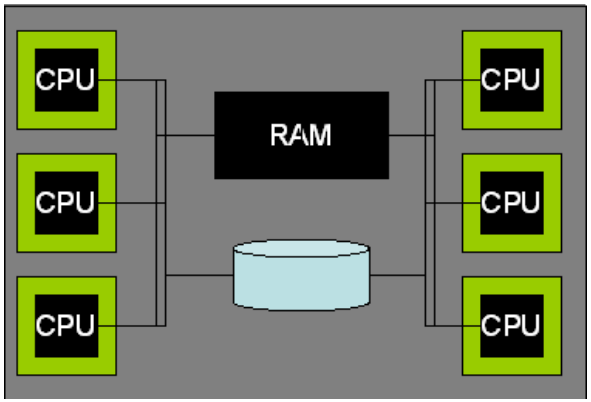


Fig. 3.1 Shared-memory architecture.

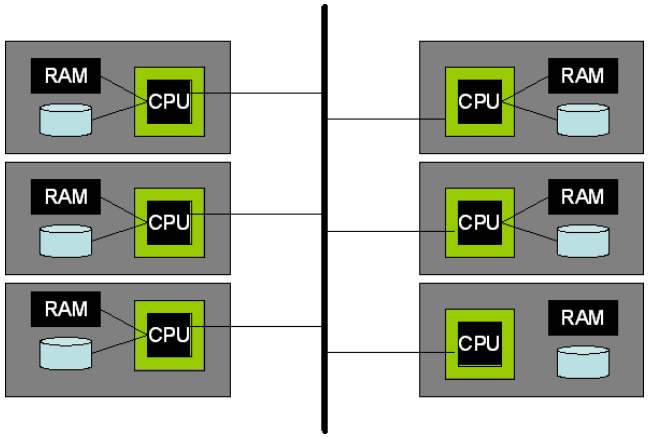


Fig. 3.2 Shared-nothing architecture.

Parallel Architectures

- Shared-disk (e.g., through use of Storage Area Networks)
  - Somewhat easier to administer, but requires specialized hardware
  - Main difference between this and shared-nothing is primarily the retrieval costs
- Non-uniform Memory Access (NUMA)
  - Seen increasingly today with many-core systems
  - Any processor can access any other processor's memory, but the costs vary

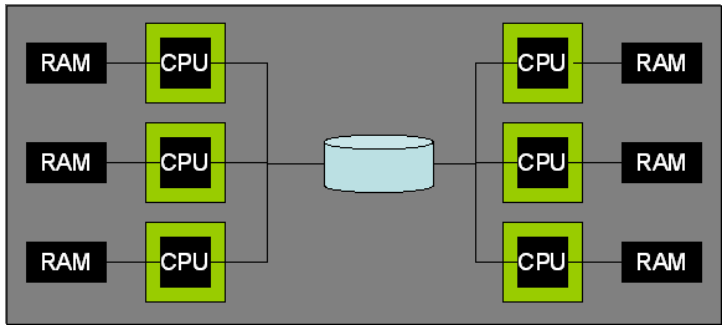
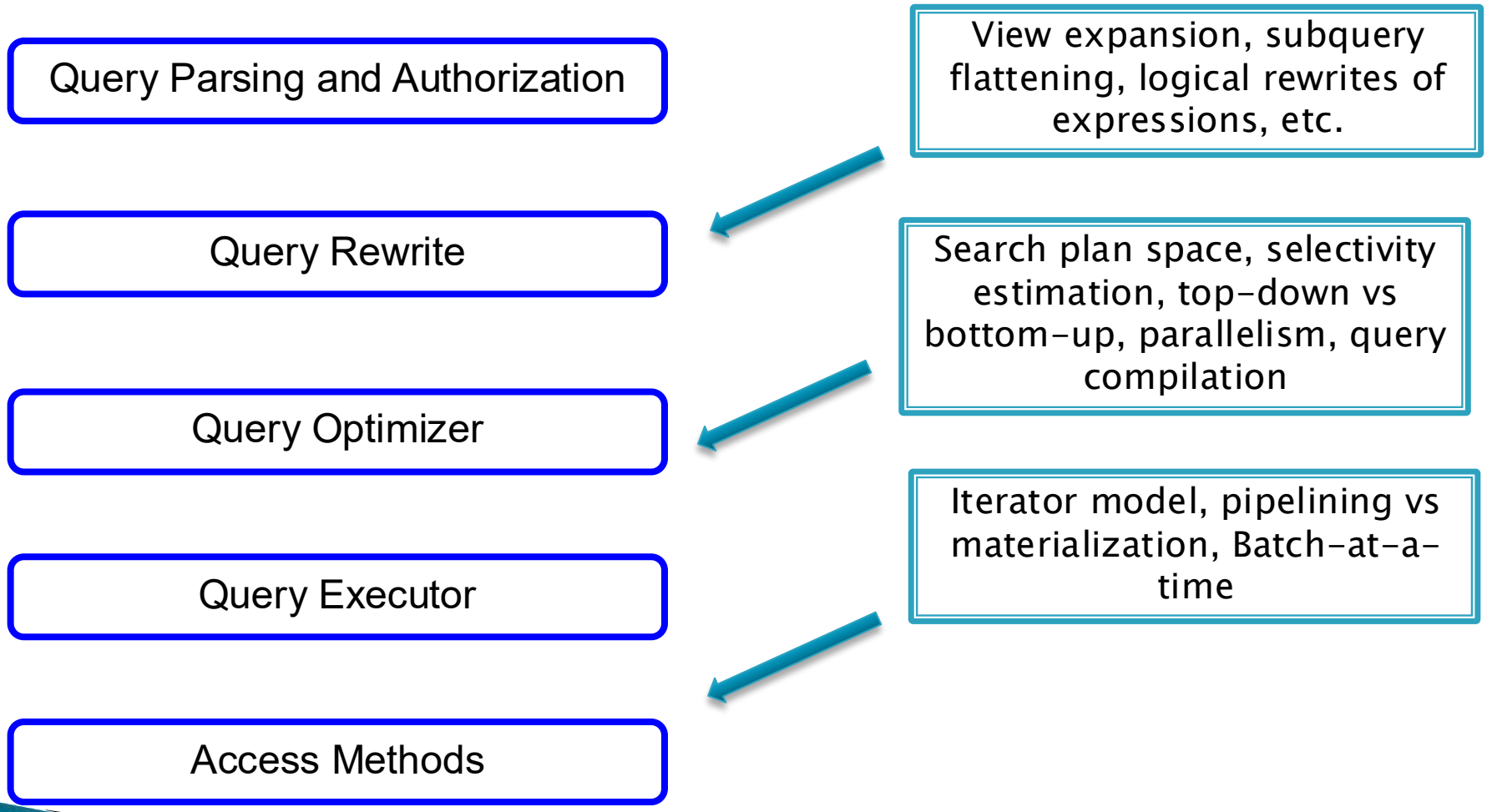


Fig. 3.3 Shared-disk architecture.

Relational Query Processor



Data Warehouses

- Widely used today for large-scale analytics
- Use specialized index structures (like bitmap indexes)
- Bulk uploads of batches of data
- Materialized Views
- OLAP and Data Cubes
- Specialized optimization techniques
  - Snowflake schemas are very common
  - Often use techniques like Bloom Filters or bitmap based operations
  - Use Columnar Storage today

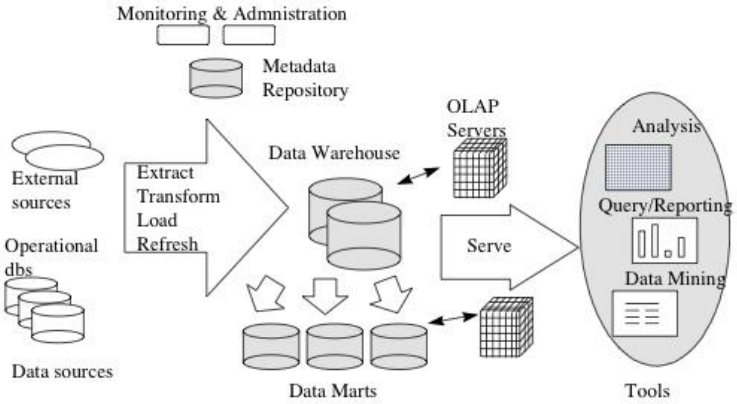


Figure 1. Data Warehousing Architecture

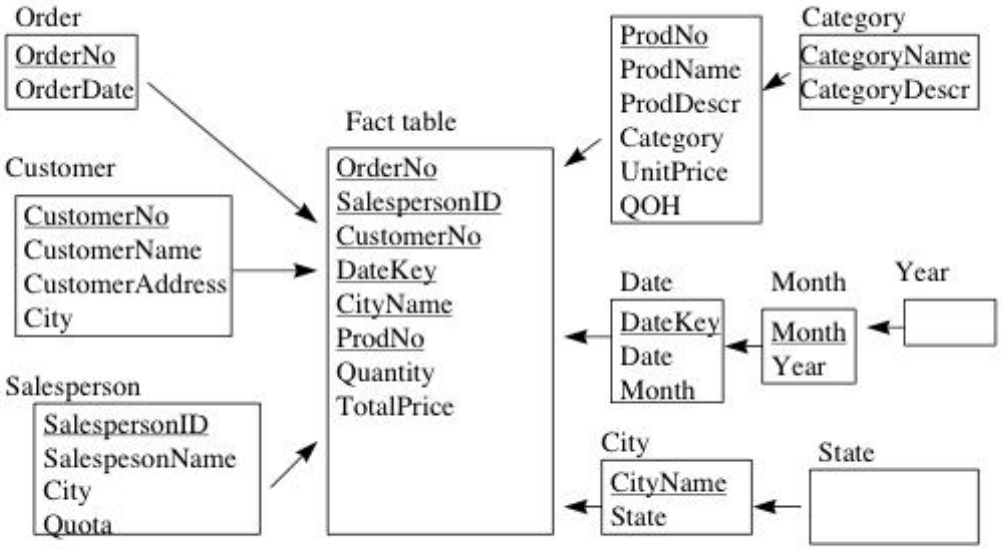


Figure 4. A Snowflake Schema.

Storage Management

- Databases need to be able to control:
  - Where data is physically stored on the storage devices, especially what is sequentially stored (i.e., spatial locality)
    - To reduce/estimate costs of operations
  - What is in memory vs not in memory (temporal locality)
    - To optimize query execution
  - How is memory managed
    - To avoid double copying of data
  - In which order data is written out of volatile storage (memory) into non-volatile storage (disks/SSDs)
    - For guaranteeing correctness in presence of failures

- Operating systems often get in the way
  - Databases often allocate a large file on disk and manage spatial locality themselves (no guarantees that the file is sequential though)
  - Use memory mapping to reduce double copying within memory
  - And many other tricks to get around OS restrictions…

Transactions

- ACID properties
  - Atomicity, Isolation, and Durability are database guarantees – Consistency is typically a programmer guarantee
- Serializability: A notion of "correctness" of concurrent transactions
  - Standard approaches: Strict 2-phase Locking, Multi-version Concurrency Control, Optimistic Concurrency Control
  - A lot of work in the last 15 years – MVCC probably considered the best option today
- Difference between "locking" and "latching"
  - Latches are more low-level, basically synchronization primitives
  - Locks are logical and taken on, e.g., relations/tuples/objects, etc.
- Isolation Levels
  - From the early days, databases supported looser definitions of consistency
  - Not easy to formalize
- Recovery
  - Traditionally done through "logging", i.e., keep a record of all updates and use it for undoing bad changes, and redoing good changes

Shared Components

- Catalog Manager (more appropriately today: “Metadata” Manager)
  - Usually stored as special system tables
  - Pulled into memory at the start for efficiency, into special data structures
- Memory Allocator
  - Need to be very careful with allocating new chunks of memory
  - PostgreSQL query processor basically pre-allocates everything and reuses all the memory
- Disk Management Subsystems
  - Many different storage devices widely used (e.g., RAID)
  - Need to support a uniform interface on top (through abstractions)
  - Makes optimization harder
- Replication Services
- Administration, Monitoring, Utilities

Recap, and Next Steps

- Read the “Architecture” paper, and raise any questions/clarification issues
- Although outdated, this will form the basis on which the rest of the semester builds up
  - First written assignment will cover some of these topics as well
- Next two weeks:
  - Different data models/query languages/programming frameworks
  - Will ignore the implementation issues in the papers