


# CMSC 724: Database Management Systems

## Introduction/Background

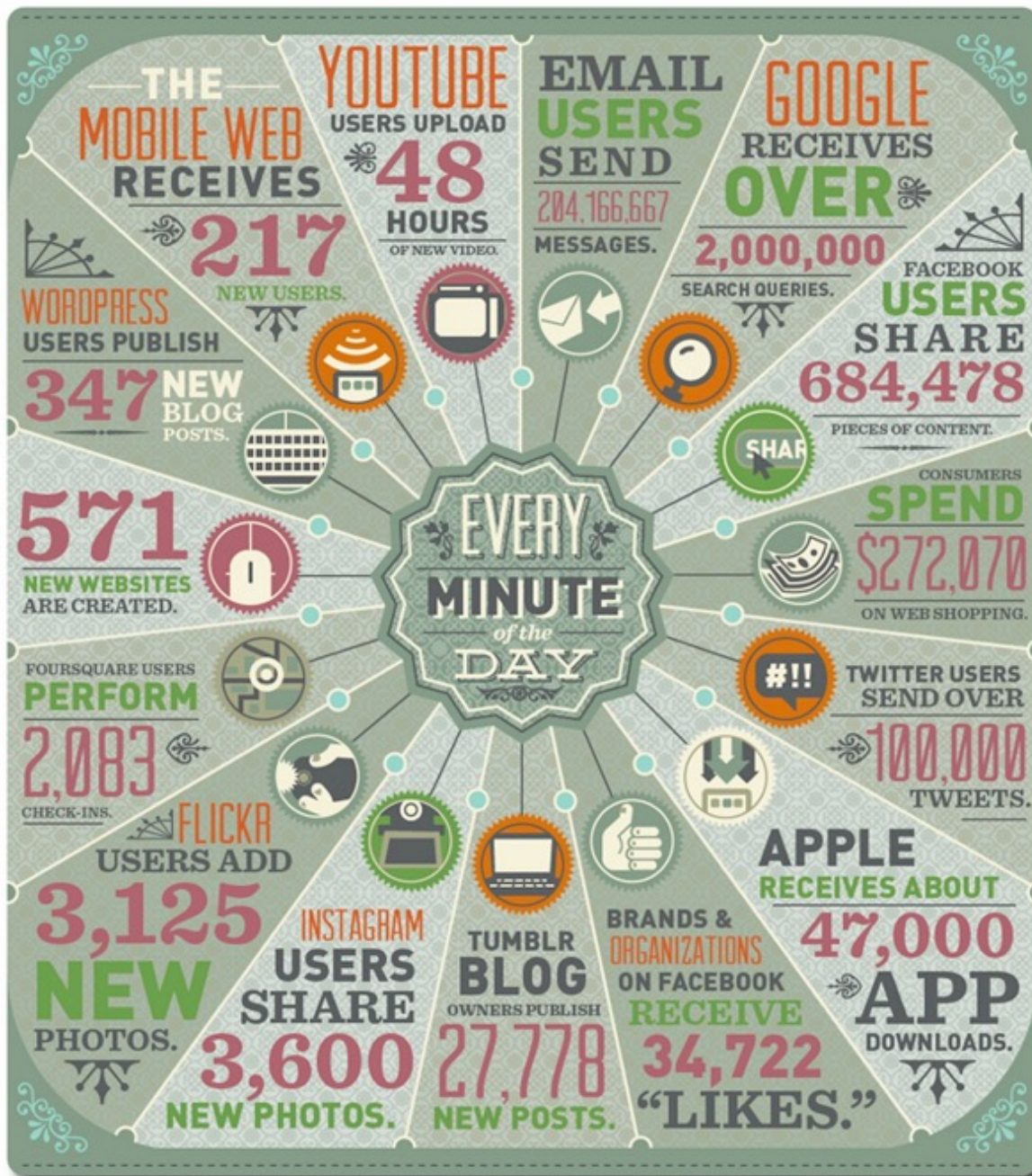
Instructor: Amol Deshpande  
amol@cs.umd.edu

# 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 Study Databases?

- ▶ There is a **\*HUGE\*** amount of data in this world
- ▶ Everywhere you see...
- ▶ Personal
  - Emails, data on your computer
- ▶ Enterprise
  - The original primary motivation
  - Banks, supermarkets, universities, airlines, phone call data etc.
- ▶ Scientific
  - Biological, astronomical
- ▶ World wide web
  - Social networks etc...





**40 ZETTABYTES**

[ 43 TRILLION GIGABYTES ]  
of data will be created by 2020, an increase of 300 times from 2005



**Volume**  
SCALE OF DATA



It's estimated that **2.5 QUINTILLION BYTES**

[ 2.3 TRILLION GIGABYTES ]  
of data are created each day



Most companies in the U.S. have at least **100 TERABYTES**  
[ 100,000 GIGABYTES ]  
of data stored

**The FOUR V's of Big Data**

From traffic patterns and music downloads to web history and medical records, data is recorded, stored, and analyzed to enable the technology and services that the world relies on every day. But what exactly is big data, and how can these massive amounts of data be used?

As a leader in the sector, IBM data scientists break big data into four dimensions: **Volume, Velocity, Variety and Veracity**

Depending on the industry and organization, big data encompasses information from multiple internal and external sources such as transactions, social media, enterprise content, sensors and mobile devices. Companies can leverage data to adapt their products and services to better meet customer needs, optimize operations and infrastructure, and find new sources of revenue.

By 2015 **4.4 MILLION IT JOBS** will be created globally to support big data, with 1.9 million in the United States



As of 2011, the global size of data in healthcare was estimated to be

**150 EXABYTES**

[ 161 BILLION GIGABYTES ]



**30 BILLION PIECES OF CONTENT**

are shared on Facebook every month



By 2014, it's anticipated there will be **420 MILLION WEARABLE, WIRELESS HEALTH MONITORS**

**4 BILLION+ HOURS OF VIDEO** are watched on YouTube each month

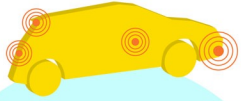


**400 MILLION TWEETS** are sent per day by about 200 million monthly active users



**Variety**  
DIFFERENT FORMS OF DATA

The New York Stock Exchange captures **1 TB OF TRADE INFORMATION** during each trading session



Modern cars have close to **100 SENSORS** that monitor items such as fuel level and tire pressure

**Velocity**  
ANALYSIS OF STREAMING DATA

By 2016, it is projected there will be **18.9 BILLION NETWORK CONNECTIONS** - almost 2.5 connections per person on earth



**1 IN 3 BUSINESS LEADERS**

don't trust the information they use to make decisions



Poor data quality costs the US economy around **\$3.1 TRILLION A YEAR**



in one survey were unsure of how much of their data was inaccurate

**Veracity**  
UNCERTAINTY OF DATA

Sources: McKinsey Global Institute, Twitter, Cisco, Gartner, EMC, SAS, IBM, MEPTec, QAS

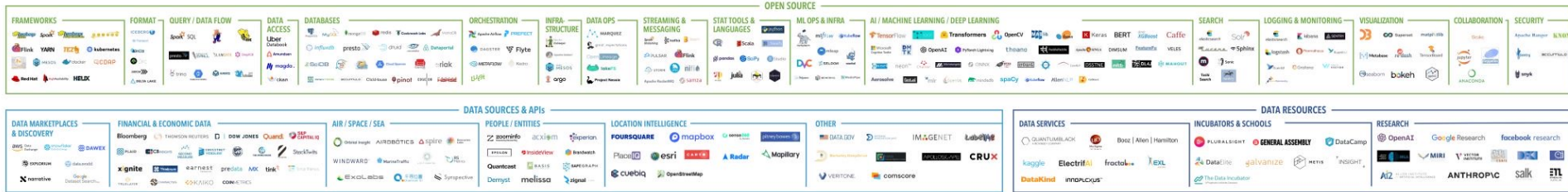
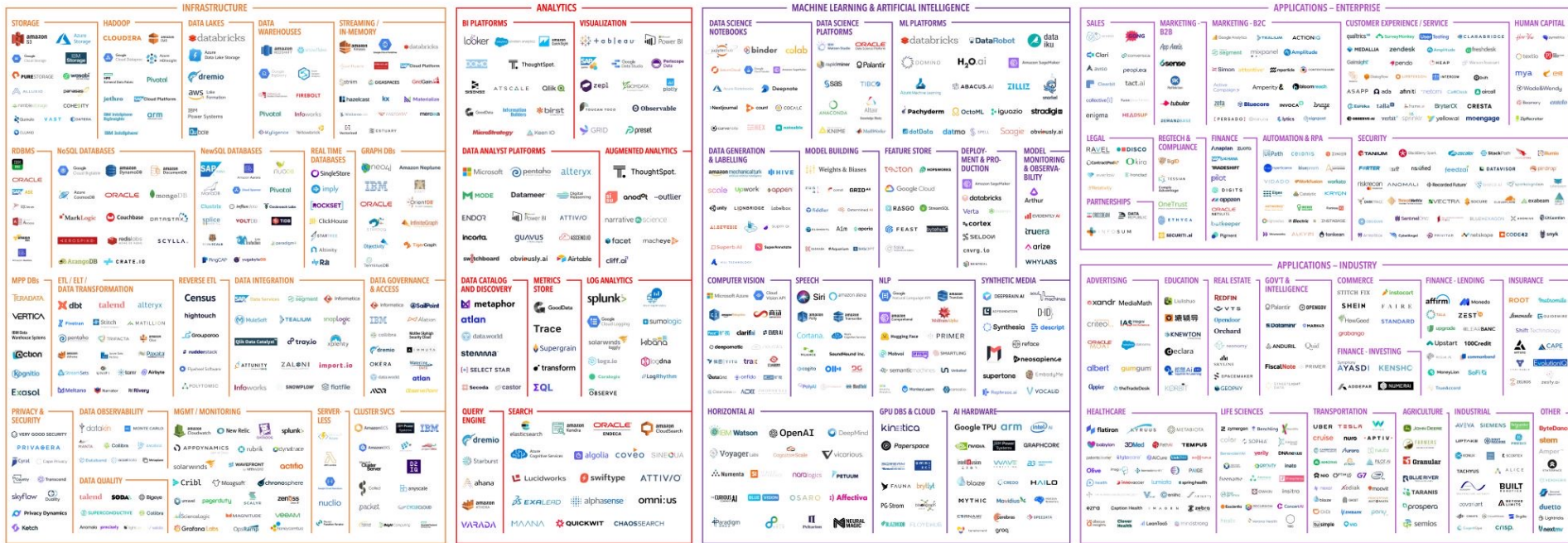


# Data Management Systems

Data management systems at the center of most of the new innovative technologies

An incredible amount of ongoing work in building new types of systems

MACHINE LEARNING, ARTIFICIAL INTELLIGENCE, AND DATA (MAD) LANDSCAPE 2021





# INFRASTRUCTURE

## STORAGE

**amazon S3** **Azure Storage**  
**Google Cloud Storage** **IBM Storage**  
**PURESTORAGE** **wasabi**  
**ALLUXIO** **panasas**  
**nimblestorage** **COHESITY**  
**Qumulo** **VAST** **DATERA**  
**CLUMIO**

## HADOOP

**CLOUDERA** **amazon EMR**  
**Google Cloud Dataproc** **Azure HDInsight**  
**HPE** **Pivotal**  
**EMERALD Data Fabric**  
**jethro** **Cloud Platform**  
**IBM InfoSphere BigInsights** **ARM**  
**IBM InfoSphere**

## DATA LAKES

**databricks**  
**Azure Data Lake Storage**  
**dremio**  
**aws Lake Formation**  
**IBM Power Systems**  
**Duoble**

## DATA WAREHOUSES

**amazon REDSHIFT** **snowflake**  
**Google BigQuery** **Redshift Spectrum**  
**ORACLE Exadata Cloud Service** **FIREBOLT**  
**Pivotal** **Infoworks**  
**Kyligence** **Yellowbrick**

## STREAMING / IN-MEMORY

**amazon Kinesis** **Google Cloud Pub/Sub** **databricks**  
**confluent** **EMRFS** **Cloud Platform**  
**stream** **GIGASPACE** **GridGain**  
**hazelcast** **lx** **Materialize**  
**Wavefront** **FASTORBIT** **meroka**  
**Vectorize** **ESTUARY**

## RDBMS

**IBM DB2**  
**ORACLE**  
**Microsoft ASE**  
**SQL Server**  
**Microsoft Access**  
**amazon RDS**  
**Amazon Aurora**

## NoSQL DATABASES

**Google Cloud Bigtable** **amazon DynamoDB** **amazon DocumentDB**  
**Azure CosmosDB** **ORACLE** **mongoDB**  
**MarkLogic** **Couchbase** **DATASTAX**  
**EROSPIKE** **redislabs** **SCYLLA**  
**ArangoDB** **CRATE.IO**

## NewSQL DATABASES

**SAP HANA** **Amazon Aurora** **nuodb**  
**Microsoft SQL Server** **Cloud Spanner** **Pivotal**  
**Clustrix** **influxdata** **Cockroach Labs**  
**splice** **VOLTD** **TIDB**  
**11vSCALE** **Trifolium** **paradigm**  
**PingCAP** **yugabyteDB**

## REAL TIME DATABASES

**SingleStore**  
**imply**  
**ROCKSET**  
**ClickHouse**  
**STARTREE**  
**Altnity**  
**Rail**

## GRAPH DBs

**neo4j** **Amazon Neptune**  
**IBM**  
**ORACLE** **OrientDB**  
**STARDOG** **InfiniteGraph**  
**Objectivity** **TigerGraph**  
**TerminusDB**

## MPP DBs

**TERADATA**  
**VERTICA**  
**IBM Data Warehouse Systems**  
**Qaction**  
**Kognitio**  
**Exasol**

## ETL / ELT / DATA TRANSFORMATION

**dbt** **talend** **alteryx**  
**Fivetran** **Stitch** **MATILLION**  
**pentaho** **TRIFACTA** **amazon Glue**  
**amazon Athena** **Azure Data Factory** **Paxata**  
**StreamSets** **UNIFI** **tamr** **Airbyte**  
**Meltano** **Narrator** **Ri Rivery**

## REVERSE ETL

**Census**  
**hightouch**  
**Grouparoo**  
**rudderstack**  
**Flywheel Software**  
**POLYTOMIC**

## DATA INTEGRATION

**SAP Data Services** **segment** **Informatica**  
**MuleSoft** **TEALIUM** **snapLogic**  
**Qlik Data Catalyst** **tray.io** **xplenty**  
**ATTUNITY** **ZALONI** **import.io**  
**Infoworks** **SNOWPLOW** **flatfile**

## DATA GOVERNANCE & ACCESS

**Informatica** **SailPoint**  
**IBM** **Alation**  
**collibra** **McAfee Skyhigh Security Cloud**  
**dremio** **PHUTA**  
**OKERA** **Waterline Data**  
**data.world** **atlan**  
**ANR** **ObservePoint**

## PRIVACY & SECURITY

**VERY GOOD SECURITY**  
**PRIVAERA**  
**Cyral** **Cape Privacy**  
**Country** **Transcend**  
**skyflow** **Duality**  
**Privacy Dynamics**  
**Ketch**

## DATA OBSERVABILITY

**datakin** **MONTE CARLO**  
**MANTA** **Collibra** **datafold**  
**Databand** **acceldata** **Metaspine**

## DATA QUALITY

**talend** **SODA** **Bigeye**  
**SUPERCONDUCTIVE** **Collibra**  
**Anomalo** **precisely** **lightbowl** **valido**

## MGMT / MONITORING

**amazon Cloudwatch** **New Relic** **DATADOG** **splunk**  
**APPDYNAMICS** **rubrik** **dynatrace**  
**solarwinds** **WAVEFRONT** **octifio**  
**Cribl** **Moogsoft** **chronosphere**  
**Unravel** **pagerduty** **SCALYR** **zenoss**  
**ScienceLogic** **MAGNITUDE** **VEBAM**  
**Grafana Labs** **OpsRamp** **Fineract**

## SERVER-LESS

**Azure**  
**IBM Lambda**  
**Google Cloud Functions**  
**nuclio**  
**Ported Function Service**

## CLUSTER SVCS

**Amazon ECS** **IBM Power Systems** **IBM**  
**Amazon EKS** **Red Hat OpenShift**  
**Microsoft Cluster Server** **D2 IQ**  
**Colled** **anyscale**  
**packet** **CYCLECLOUD**  
**Cloud Foundry** **Bright Computing** **Microsoft Azure**

# Data Management

## **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

## **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

# 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)
- ▶ Miscellaneous Topics (2.5 weeks)
  - Versioning, Immutability, Security, Privacy

# 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


- ▶ We will cover:
  - A blend of classic papers + ongoing research (more focus on latter)
  - Reference book:
    - Readings in Database Systems, 5th edition. Mike Stonebraker and 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 Overview: Grading

- ▶ 4-5 Programming Assignments (20%)
  - Primarily on using different types of systems
- ▶ Written homeworks on the paper readings (30%)
  - 1-2 Papers per class (starting the week after next)
  - Expected to skim papers before the resp. class, and go deeper for the assignment
- ▶ Scribe notes (5%)
  - Will circulate a sign-up sheet later today
- ▶ Research project + Presentation (30%)
  - More on that later
- ▶ Final (15%)
  - Basically a slightly longer written assignment


# Course Overview: More Logistics

- ▶ Gradescope for assignments
  - ▶ Slack for communication
  - ▶ See link on the webpage
  - ▶ TA: Saptarashmi Bandyopadhyay
- 

# Databases: Major Conferences

- ▶ ACM SIGMOD (Originally SIGFIDET)
- ▶ VLDB (very large databases)
- ▶ IEEE ICDE (intl. conf. data engineering)
- ▶ EDBT (European database technology)
- ▶ PODS, ICDT
  - Theory focused
- ▶ CIDR
  - Tends to have vision/overview papers
  - I recommend browsing through 2021/2022 proceedings for ideas on class projects

# 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 !!**
- 



# Databases: A Brief History

- ▶ 1960': Enterprises start using computers – most applications had their own data store
- ▶ Data base: coined in military information systems to denote “shared data banks” by multiple apps
  - Instead of every app having a separate format and silo-ed data,
  - Define a data format, store it as a “data dictionary”, and allow general-purpose “data-base management” software to access it


# Databases: 1960's

- ▶ Birth of “hierarchical” and “network” models
    - Both allowed “connecting” records of different types explicitly
    - Network model attempted to very general and flexible (Charlie Bachman received Turing Award for this work)
    - Used COBOL language
  - ▶ IBM designed IMS hierarchical database in 1966 for Apollo space program
    - “.. more than 95 percent of the top Fortune 1000 companies use IMS to process more than 50 billion transactions a day and manage 15 million gigabytes of critical business data” (from IBM Website on IMS)
    - Predates “hard disks”
  - ▶ Both models exposed too much internal state to the users
- 

# Databases: 1970's

- ▶ Edgar F. “Ted” Codd proposed the relational model
  - Origins in set theory and logic
  - Elegant, formal model that provided almost complete “data independence”
  - High level query language (relational algebra)
  - Notion of “normal forms” – allowed one to reason about and remove redundancies
- ▶ Led to two influential projects: INGRES (Michael Stonebraker, UC Berkeley); System R (IBM)
- ▶ Also paved way for a 1977 startup called “Software Development Laboratories”
  - Didn't care about IMS/IMDS Compatibility (as IBM had to)
  - Innovated much faster

# Databases: 1970's-1990's

- ▶ 1976: Peter Chen proposed “Entity-Relationship Model”
    - Allowed high-level, conceptual modeling; easier for humans to think about
    - Continues to be widely used for that purpose
    - No real implementations – easy to convert to “relational” and use an RDBMS
    - Recent Object-relational Frameworks (like Python Django) very similar
  - ▶ 1980: Commercialization/wide-spread acceptance
    - IBM came out with DB2 in 1983
    - “SEQUEL” became the standard query language (SQL)
      - Despite significant objections
  - ▶ Late 80's: Object-oriented, object-relational databases
    - Enrich the expressive power of the relational model
    - Avoid “relational-object impedance mismatch”
    - Several other proposals for “semantic” data models
- 



# Databases: 1970's-1990's

## ▶ Late 80's, early 90's:

- Many database companies, but starting to consolidate
- Parallel database begin to emerge
- Data mining/OLAP (online analytical processing)
- Focus on client tools for application development
  - Powerbuilder (Sybase), Oracle Developer, etc.
- Client-server model becomes popular
- Postgres project at Berkeley gets an open-source fork called PostgreSQL

## ▶ Mid/Late 90's:

- Web arrives: Grown of “middleware” that connects Web Applications to Databases
  - Active server pages, Enterprise Java Beans, ColdFusion
- OLAP matures and becomes mainstream

## ▶ Core “database” research seemed like it was done... !?!

# Databases: 00's

## ▶ Early 00's to mid 00's

- A sudden boom in data warehousing/analytics
- Companies like: Aster Data, Greenplum, Vertica, Kickfire, and probably 10 others
- Significant consolidation since then
- Some key technical considerations:
  - Distributed, Shared-nothing architectures
  - Columnar data storage (instead of row data storage)
  - Focus on read-only analytics – bad write performance



# Databases: 00's

## ▶ Late 00's

- “Map-reduce”: framework for large-scale data analysis
  - Data-driven tasks that are largely “not” relational (e.g., building indexes, text analysis)
  - Pioneered by Google, open-source Hadoop system by Yahoo and others
  - An entire ecosystem built around it since then
- Key-value stores:
  - Frameworks for “scale-first” data management
  - Not as concerned by ACID properties
  - Simple get/put interfaces needed (i.e., no support for schemas/complex queries)
- Document/graph/... databases:
  - motivated by faster app development, conceptual models closer to the app developers
  - Web app developers work with JSON mostly



APACHE  
HBASE

amazon  
DynamoDB



 mongoDB®

# Databases: 10's

- ▶ Database industry/researchers react to the last decade
- ▶ “NewSQL”: scalability without giving up ACID by rearchitecting/redesigning
  - Support for parallel environments from the ground up
  - Exploit large memories and SSDs properly
  - Hybrid systems that support both transactions and analytics (HTAP)
    - Largely not considered a viable idea any more
    - Can't really get to the same performance as specialized systems



Many many more....

# Databases: 10's

- ▶ Cloud-hosted “Database-as-a-service” offerings
  - Both Data Warehouses (Snowflake, Amazon Redshift), and OLTP (Amazon Aurora, Microsoft CosmosDB)
  - Very big market today



## Snowflake On The Path To \$10B In Product Revenue

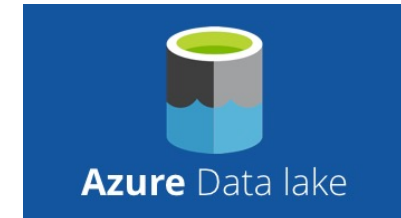
Aug. 30, 2021 6:12 PM ET | **Snowflake Inc. (SNOW)** | 9 Comments | 7 Likes

# Databases: 10's

- ▶ “Data Lakes” slowly becoming popular
  - Basically a distributed file system that is shared by everyone
  - Data processing engines read and write directly from the distributed storage
  - Typically use a columnar storage format (e.g., Parquet)
  - “Data Lakehouses” try to offer systematic search/management on top



databricks




presto 




CLUDERA

# Databases: 10's


- ▶ More specialized systems for specific types of data
    - Clear short-term benefits in performance and deployment speed
    - However, quite a few longer-term issues in having silo-ed data
  - ▶ Graph Databases
    - Neo4j, Memgraph, TigerGraph, Dgraph, TerminusDB, and many others
  - ▶ Time-series Databases
    - TimescaleDB, InfluxDB, Clickhouse, ...
  - ▶ Multi-model DBMSs
    - Support several different models (e.g., RethinkDB)
  - ▶ Databases for Machine Learning
    - e.g., frameworks to support ML on data in databases
  - ▶ Machine learning for Databases
    - e.g., improve Query Optimization through use of deep learning
- 

# 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
- 



# 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

- ▶ Massively successful for *highly structured or semi-structured data*
  - Why ? Structure in the data (if any) can be exploited for ease of use and efficiency

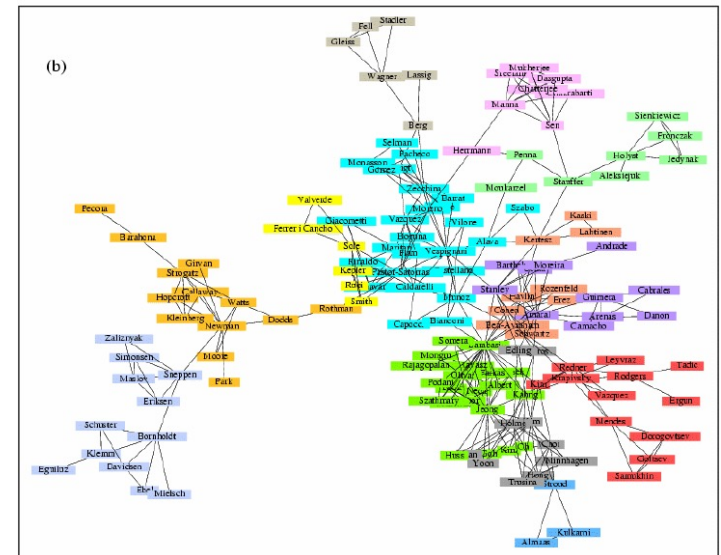
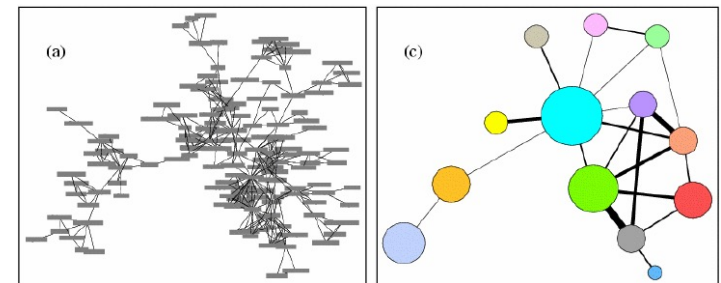
Account		
bname	acct_no	balance
Downtown	A-101	500
Mianus	A-215	700
Perry	A-102	400
R.H	A-305	350

Tabular/relational data

XML/JSON (semi-structured)

```

<Symbol>List</Symbol>
<Function>
  <Symbol>List</Symbol>
  <Symbol>Automatic</Symbol>
  <Number>4.</Number>
</Function>
<Function>
  <Symbol>List</Symbol>
  <Symbol>Automatic</Symbol>
  <Number>6.</Number>
</Function>
</Function>
</Option>
</Options>
</Notebook>
    
```



Graph-structured data



# DBMSs to the Rescue

- ▶ Massively successful for *highly structured or semi-structured data*
  - Why ? Structure in the data (if any) can be exploited for ease of use and efficiency
  - How ?
  - Two Key Concepts:
    - Data Modeling: Allows reasoning about the data at a high level
      - e.g. “emails” have “sender”, “receiver”, “...”
      - Once we can describe the data, we can start “querying” it
    - Data Abstraction/Independence:
      - Layer the system so that the users/applications are insulated from the low-level details

# DBMSs to the Rescue: Data Modeling

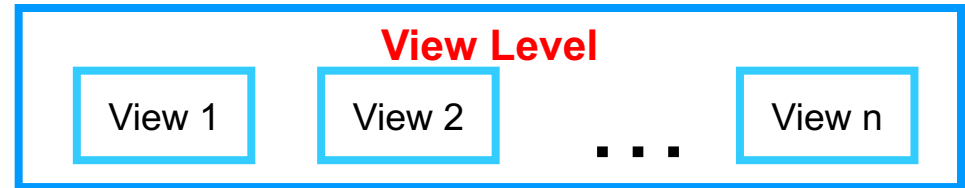
## ▶ Data modeling

- **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

# Data Abstraction/Independence

Hiding low-level details from the users of the system

**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 ?



# Relational DBMS: SQL

- ▶ **SQL** (sequel): Structured Query Language

- ▶ **Data definition (DDL)**

  - **create table** *instructor* (

    - ID*            **char(5),**

    - name*        **varchar(20),**

    - dept\_name* **varchar(20),**

    - salary*      **numeric(8,2))**

- ▶ **Data manipulation (DML)**

  - Example: Find the name of the instructor with ID 22222

    - select** *name*


    - from** *instructor*

    - where** *instructor.ID* = '22222'

# 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. (e.g., ACID guarantees)
- ▶ 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*
  - 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*

Will post a set of slides summarizing the key topics

# 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 !!**
- 

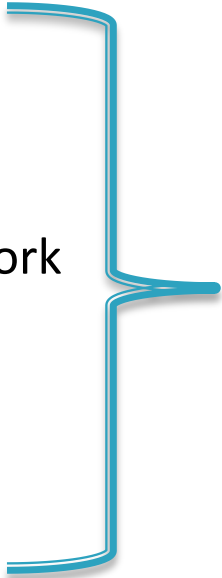
# Design Dimensions for a DMS

## ▶ 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

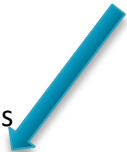
## ▶ How is data laid out on disks (at rest) and in-memory, and across machines

- 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

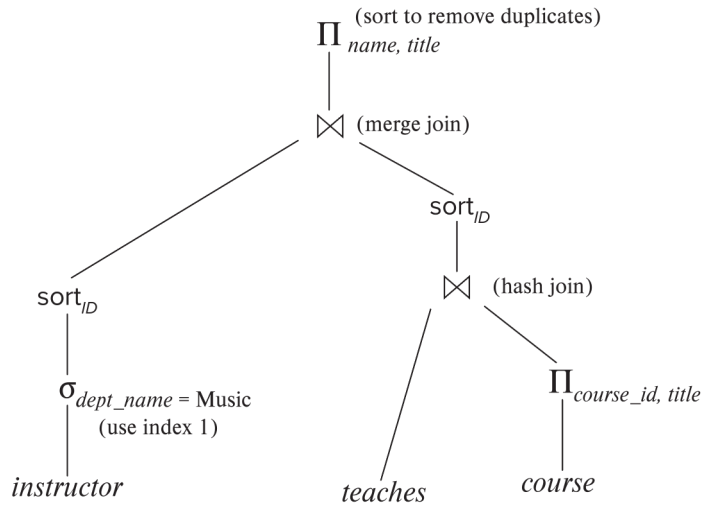
# Query Processing and Optimization

- ▶ 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

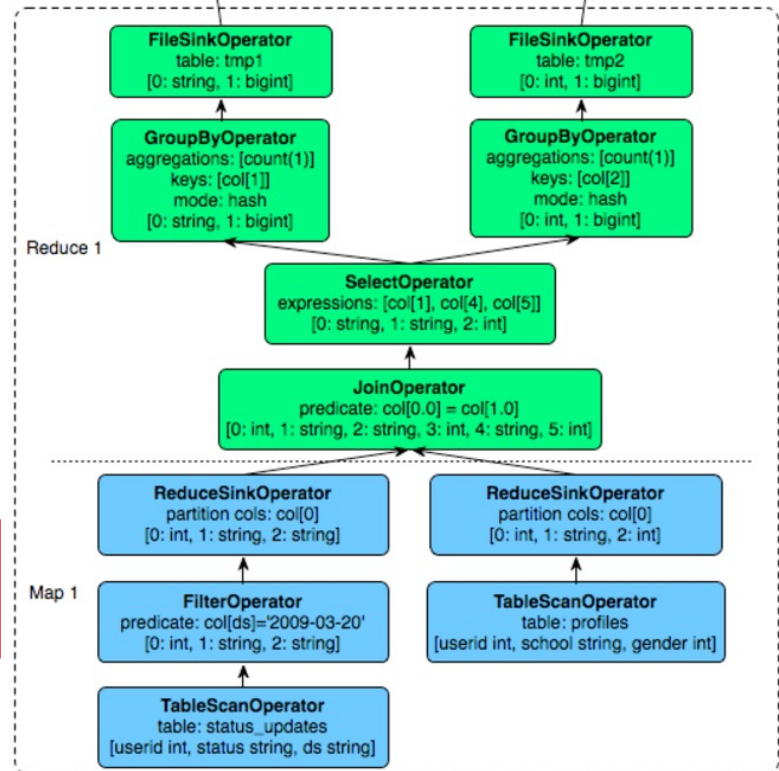
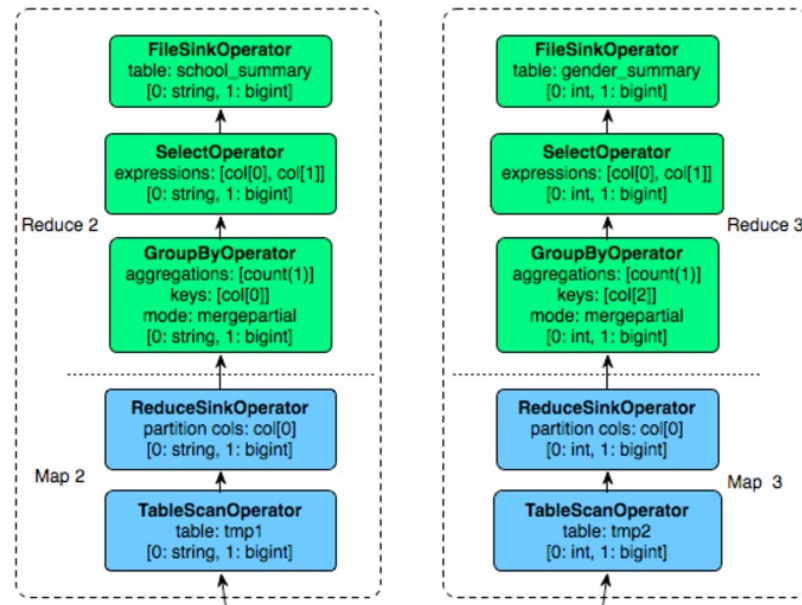
# Query Processing and Optimization

- ▶ 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...

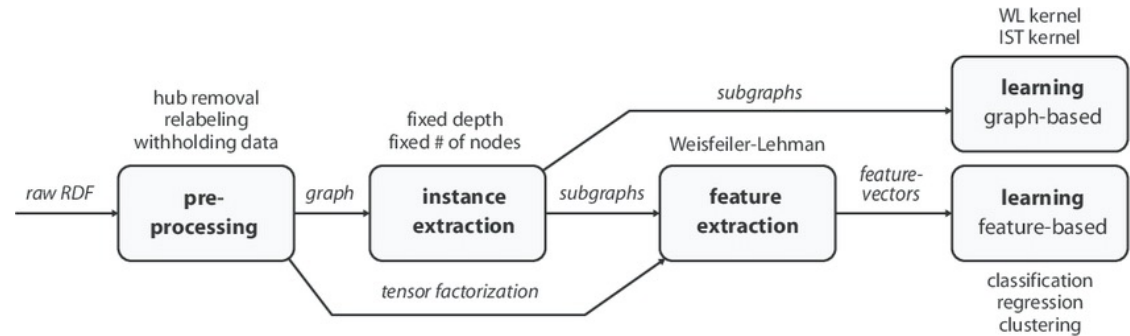


SQL "Query Plan"



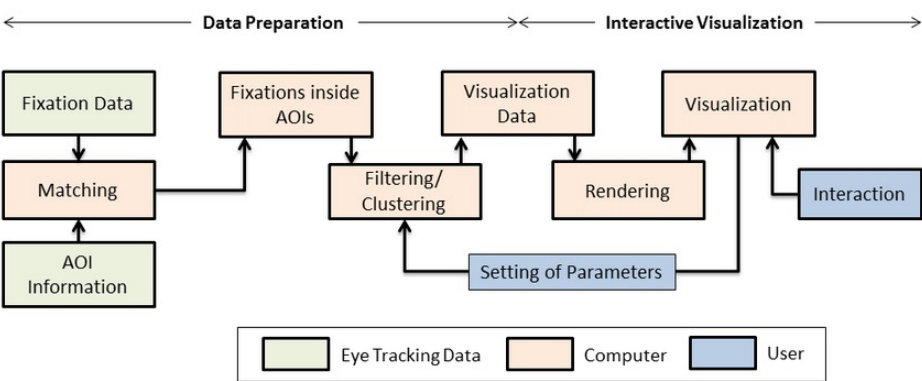
Apache Hive "Query Plan"  
(Hive is an SQL layer on top of Hadoop)

# vs ... Data Transformation Pipelines



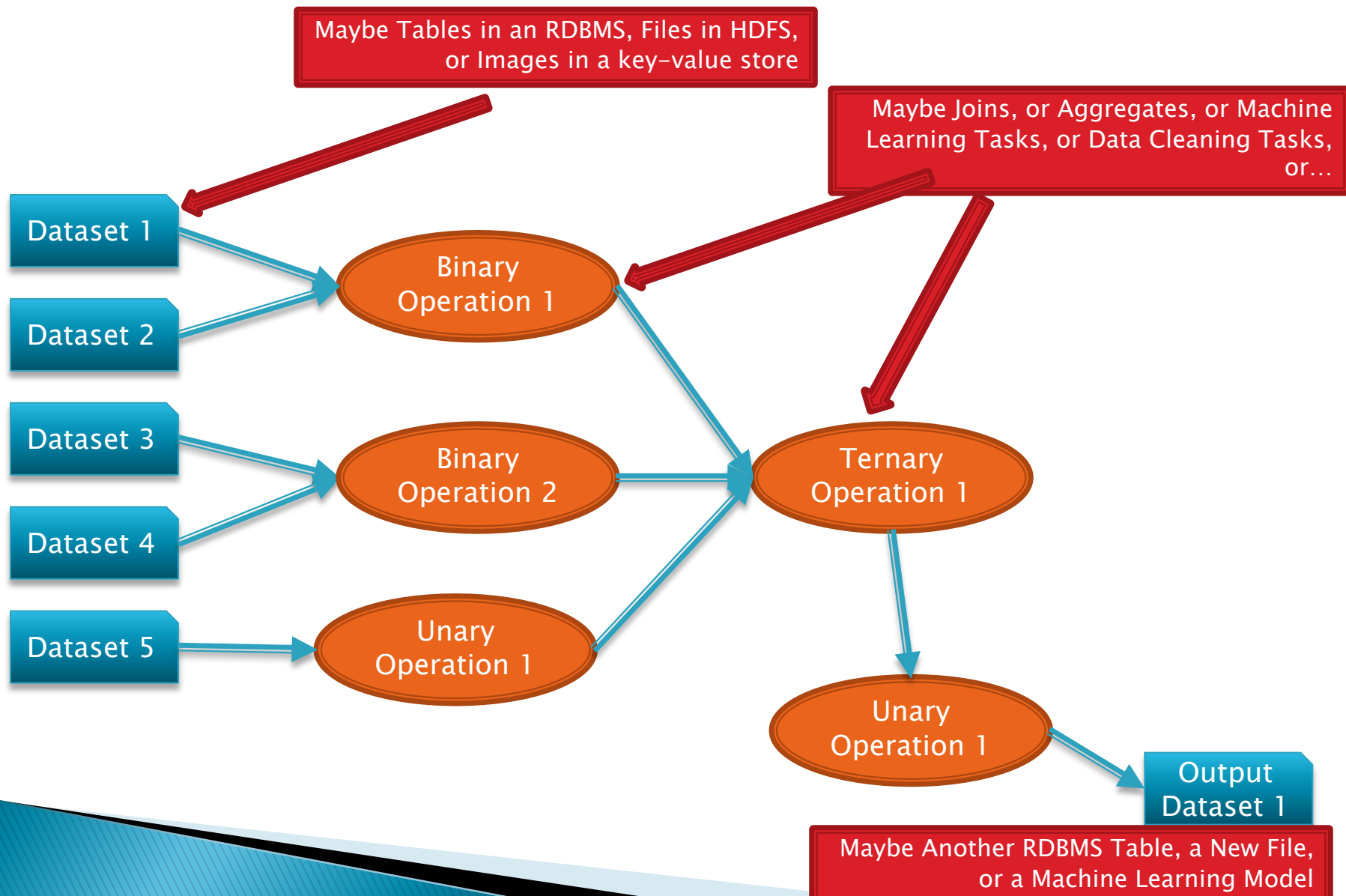
Machine Learning Pipeline

## 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
- 

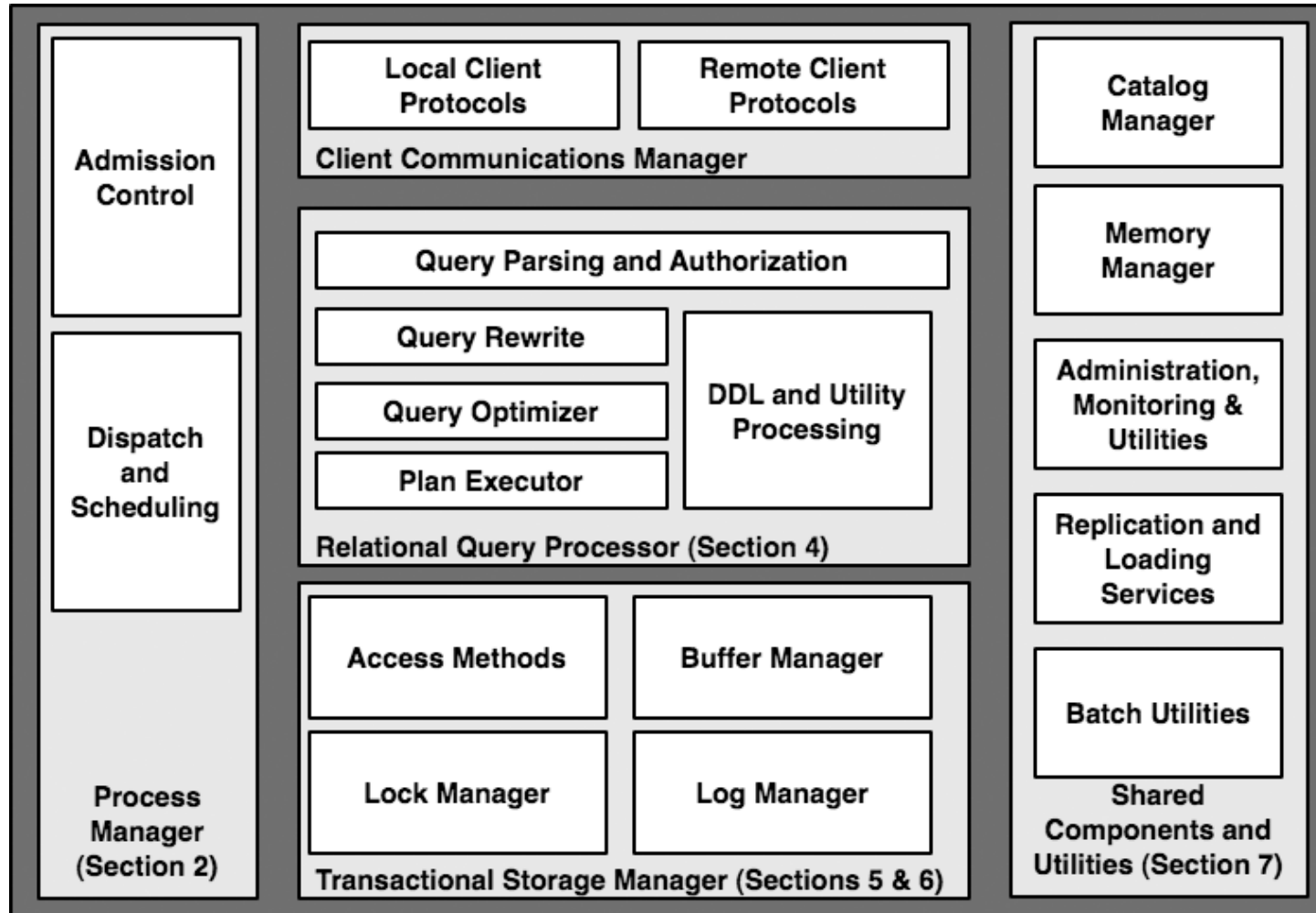
# Outline

- ▶ 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 !!**
- 

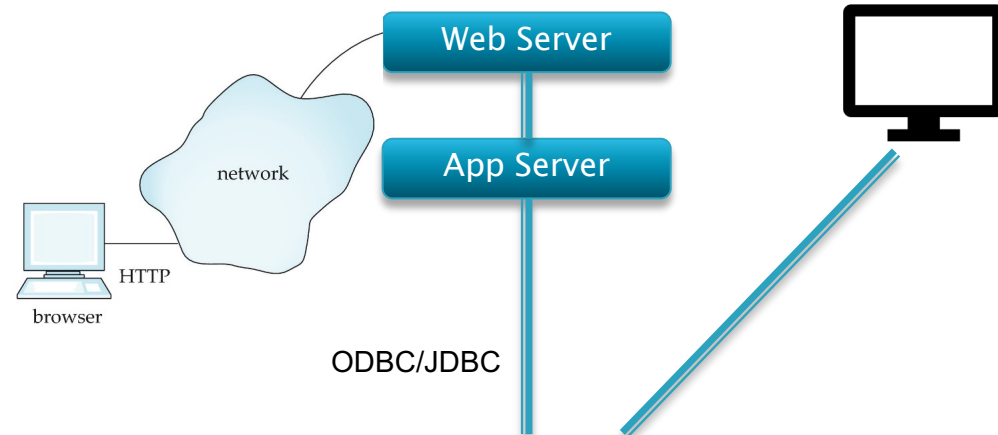
# Architecture of a Traditional DBMS

- ▶ 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

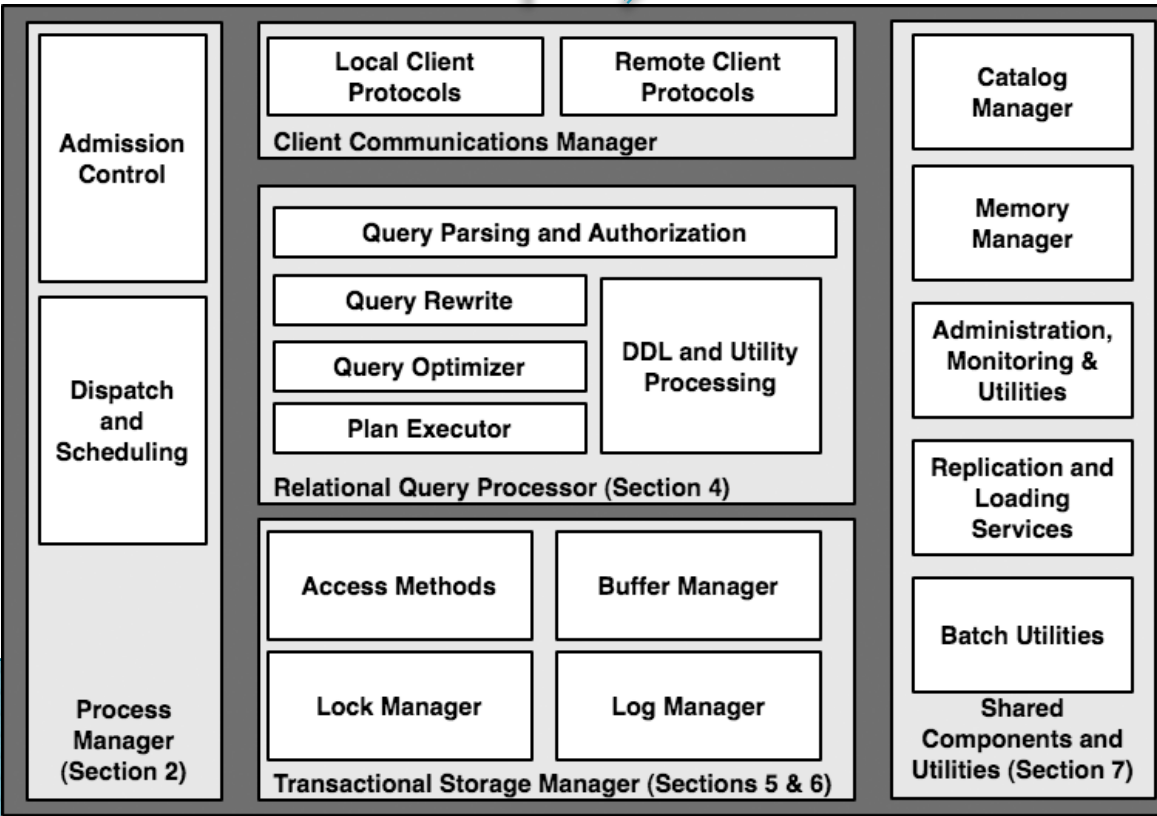
Assign a "thread of computation"

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)

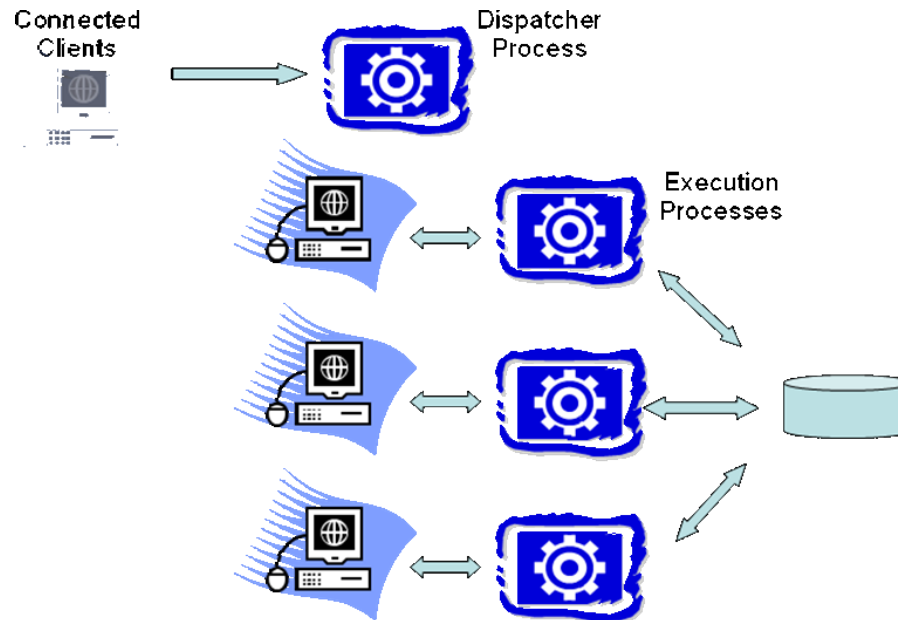


Fig. 2.1 Process per DBMS worker model: each DBMS worker is implemented as an OS process.

# 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

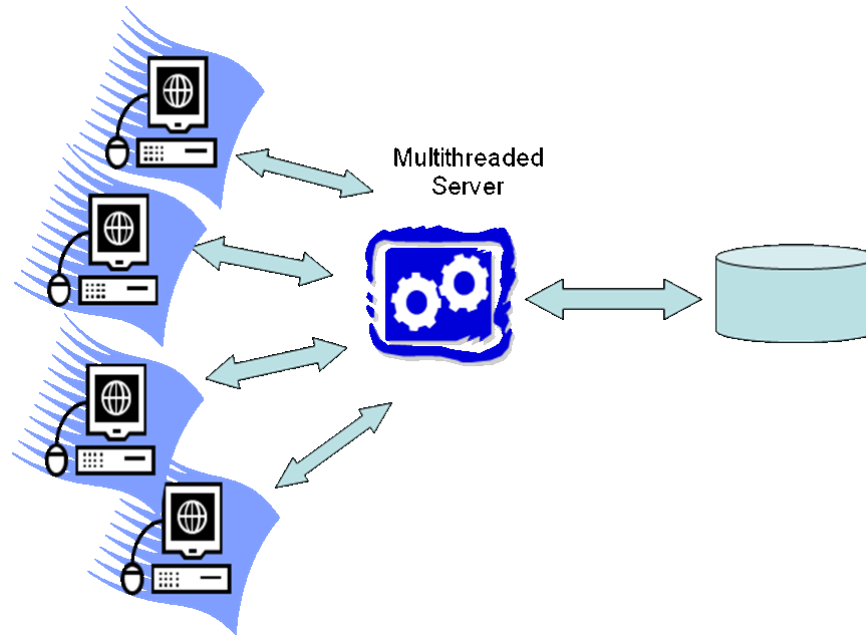


Fig. 2.2 Thread per DBMS worker model: each DBMS worker is implemented as an OS thread.

# Process (or Thread) Pools

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

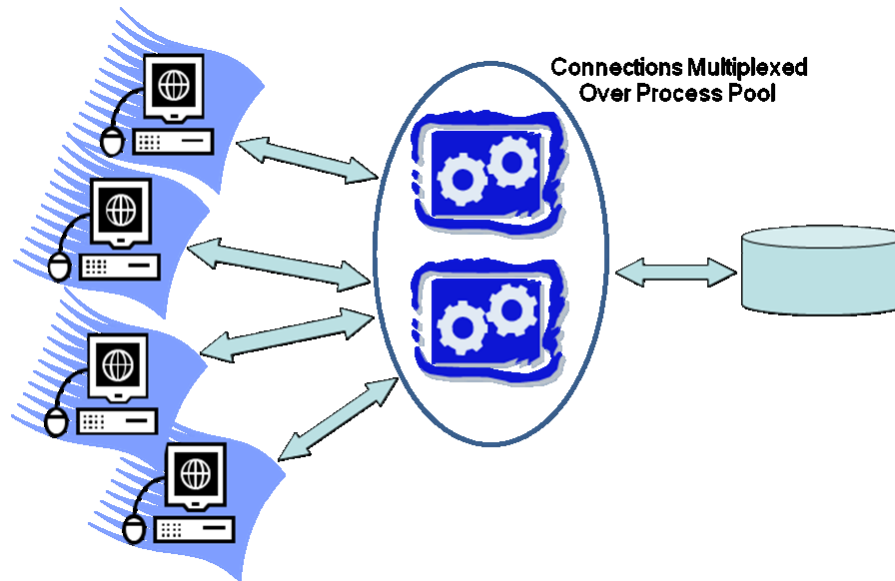




Fig. 2.3 Process Pool: each DBMS Worker is allocated to one of a pool of OS processes as work requests arrive from the Client and the process is returned to the pool once the request is processed.

# 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 be 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
- 

# 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 be 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

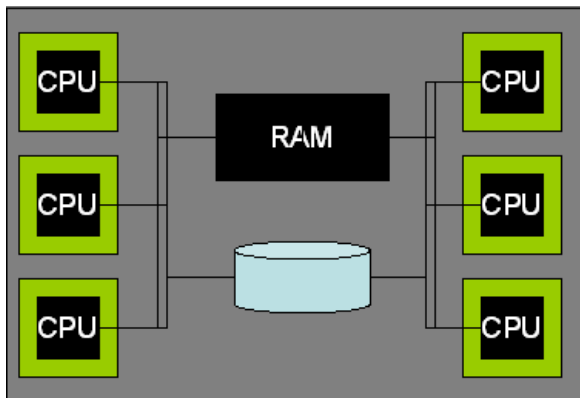


Fig. 3.1 Shared-memory architecture.

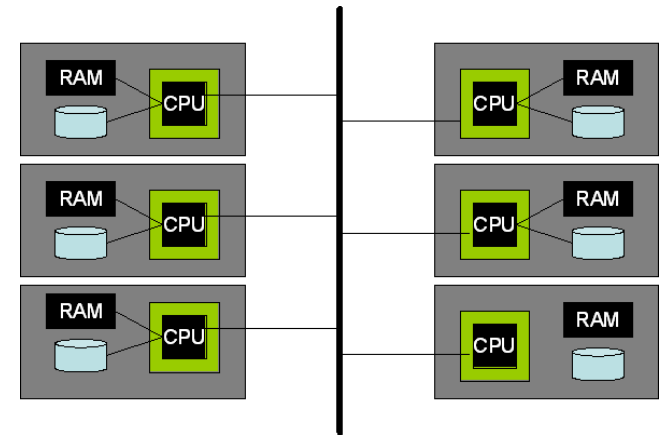


Fig. 3.2 Shared-nothing architecture.

# 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

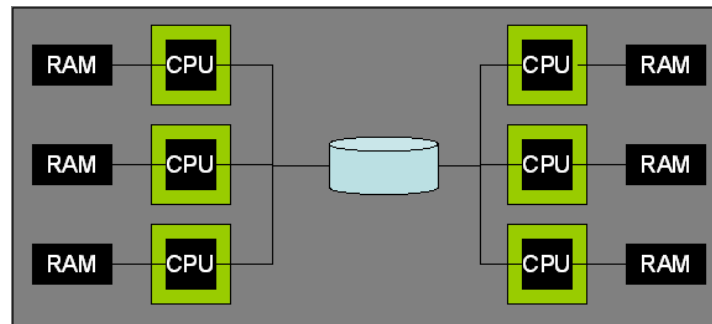
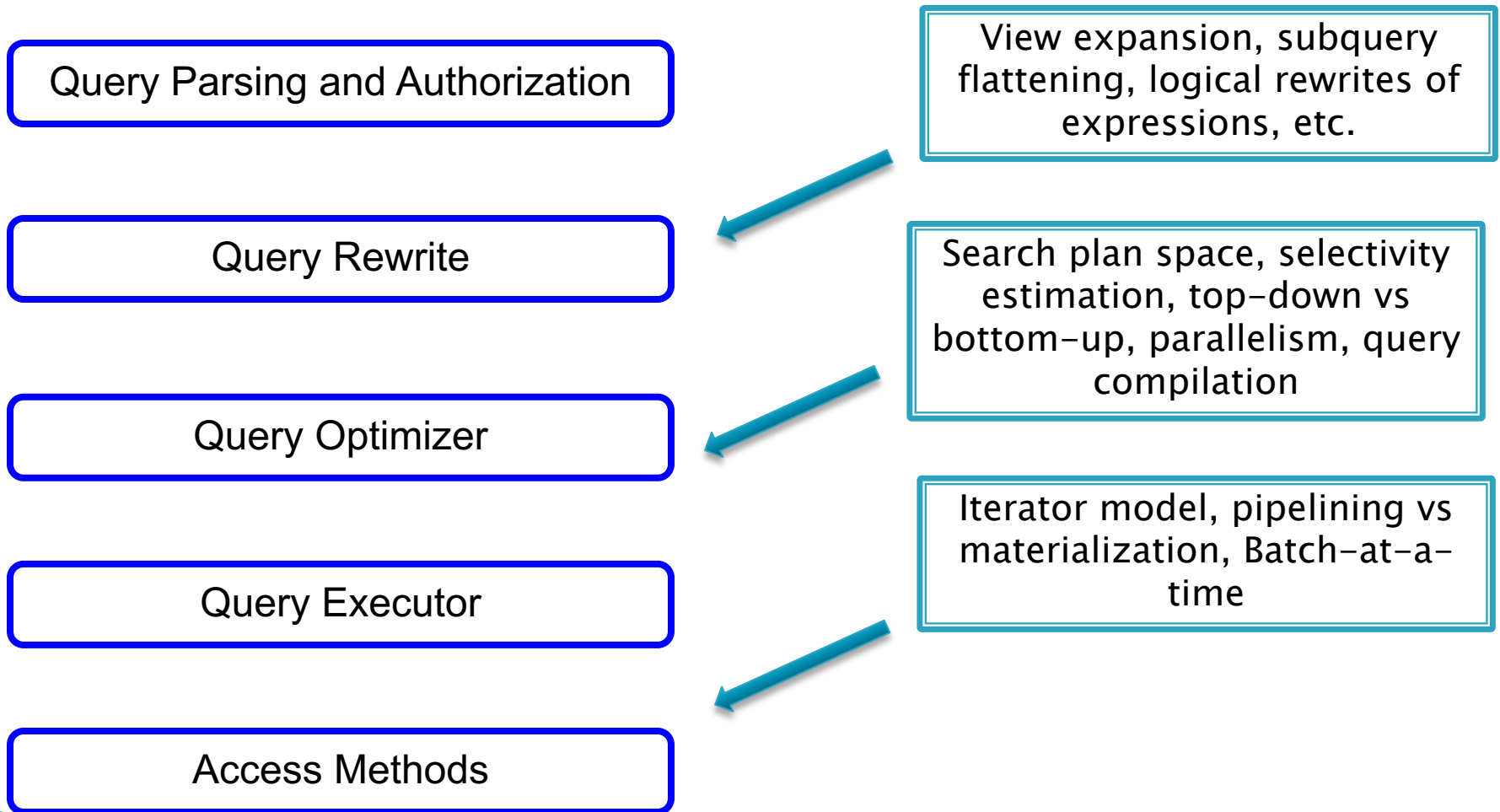


Fig. 3.3 Shared-disk architecture.

# 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

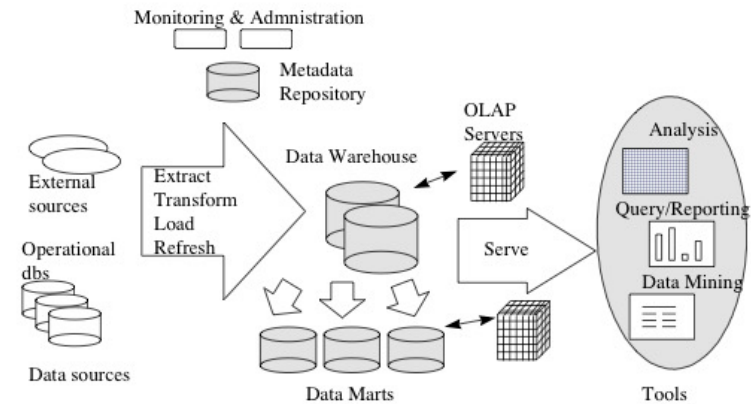


Figure 1. Data Warehousing Architecture

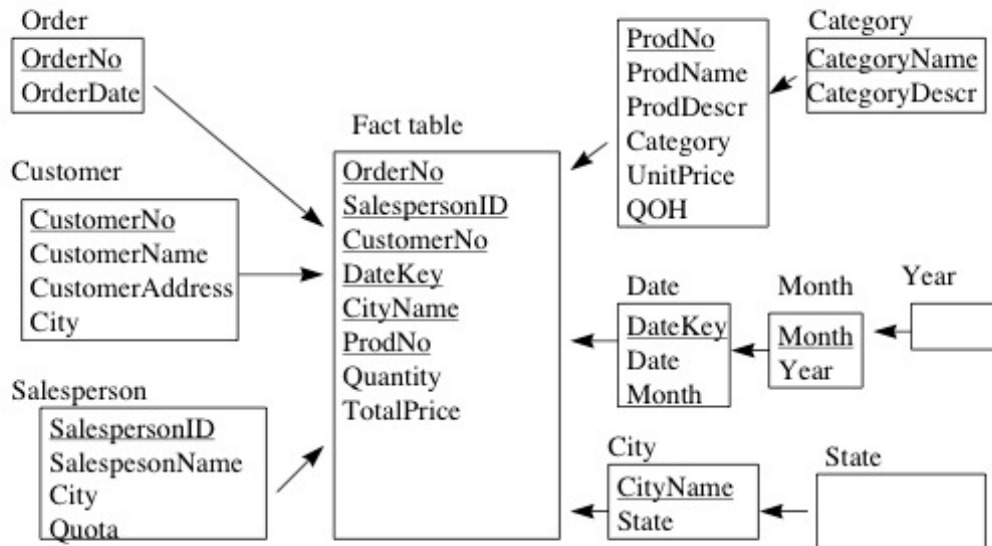


Figure 4. A Snowflake Schema.


# 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