CS 245: Principles of Data-Intensive Systems

Instructor: Matei Zaharia

cs245.stanford.edu

My Background

Berkeley UNIVERSITY OF CALIFORNIA — PhD in 2013

APACHE Spark — Open source distributed data processing framework

databricks — Data & ML platform startup

Stanford — Research in systems for ML

Outline

Why study data-intensive systems?

Course logistics

Key issues and themes

A bit of history

Why Study Data-Intensive Systems?

Most important computer applications must manage, update and query datasets

» Bank, store, fleet controller, search app, …

Data quality, quantity & timeliness becoming even more important with AI

» Machine learning = algorithms that generalize from data

What Are Data-Intensive Systems?

Relational databases: most popular type of data-intensive system (MySQL, Oracle, etc)

Many systems facing similar concerns: message queues, key-value stores, streaming systems, ML frameworks, your custom app?

Goal: learn the main issues and principles that span all data-intensive systems

Typical System Challenges

Reliability in the face of hardware crashes, bugs, bad user input, etc

Concurrency: access by multiple users

Performance: throughput, latency, etc

Access interface from many, changing apps

Security and data privacy

Practical Benefits of Studying These Systems

Learn how to select & tune data systems

Learn how to build them

Learn how to build apps that have to tackle some of these same challenges

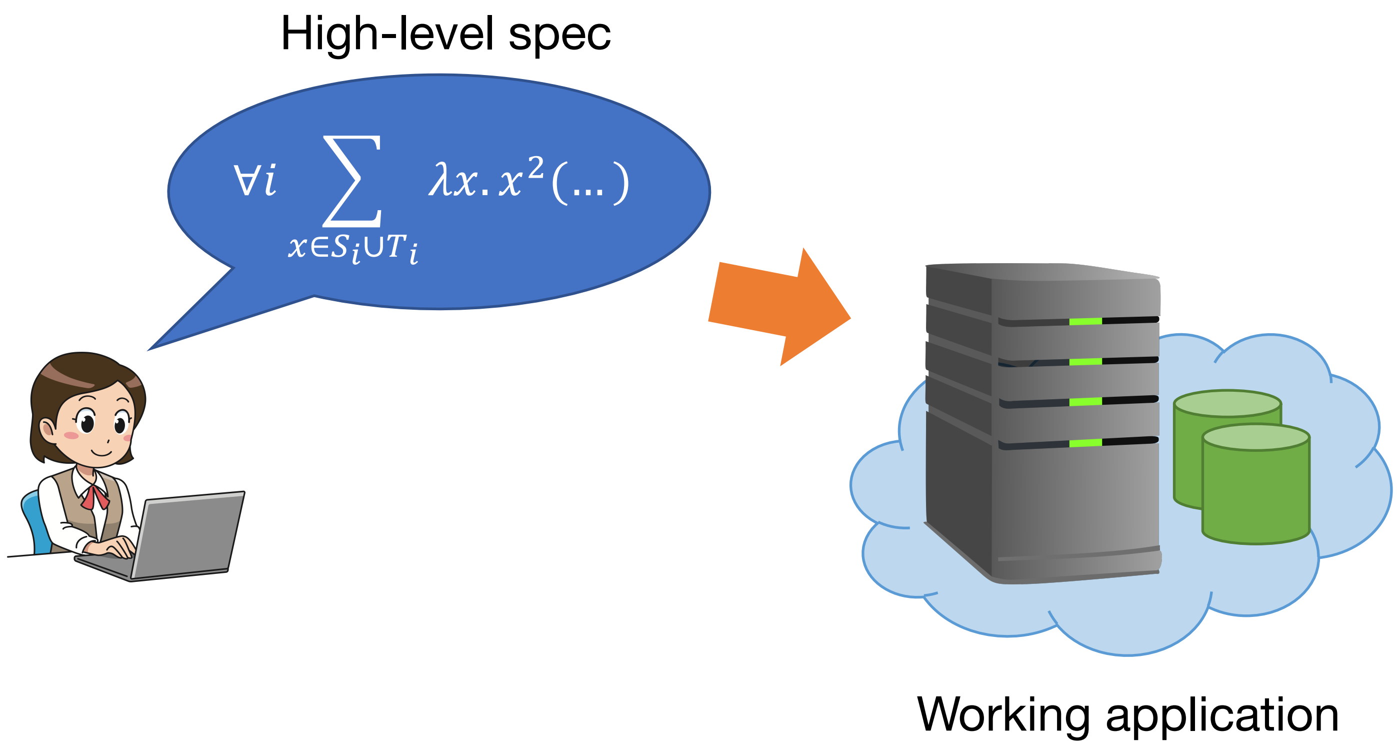
» E.g. cross-geographic-region billing app, custom search engine, etc

Scientific Interest

Interesting algorithmic and design ideas

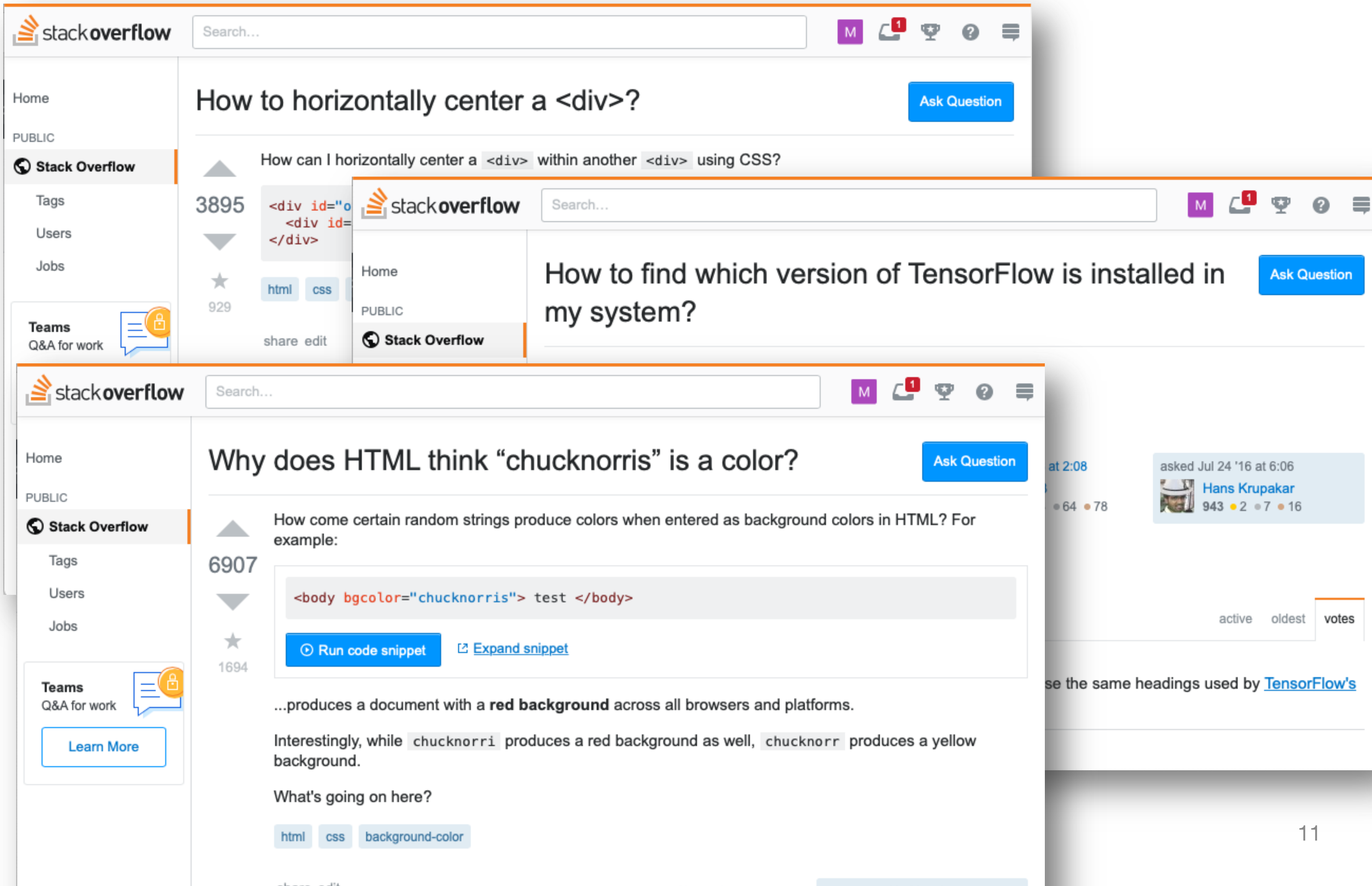
In many ways, data systems are the highest-level successful programming abstractions

Programming: The Dream



Pr

Programming: The Reality

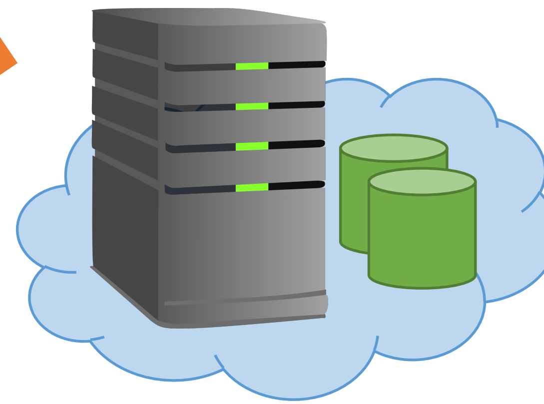


Programming with Databases

High-level spec

Relational algebra

PostgreSQL



Actually manages:
- Durability
- Concurrency
- Query optimization
- Security
- …

Outline

Why study data-intensive systems?

Course logistics

Key issues and themes

A bit of history

Teaching Assistants

Cody Coleman

Daniel Kang

Xinyi Yu

Gina Yuan

Peter Kraft

Course Format

Lectures in class

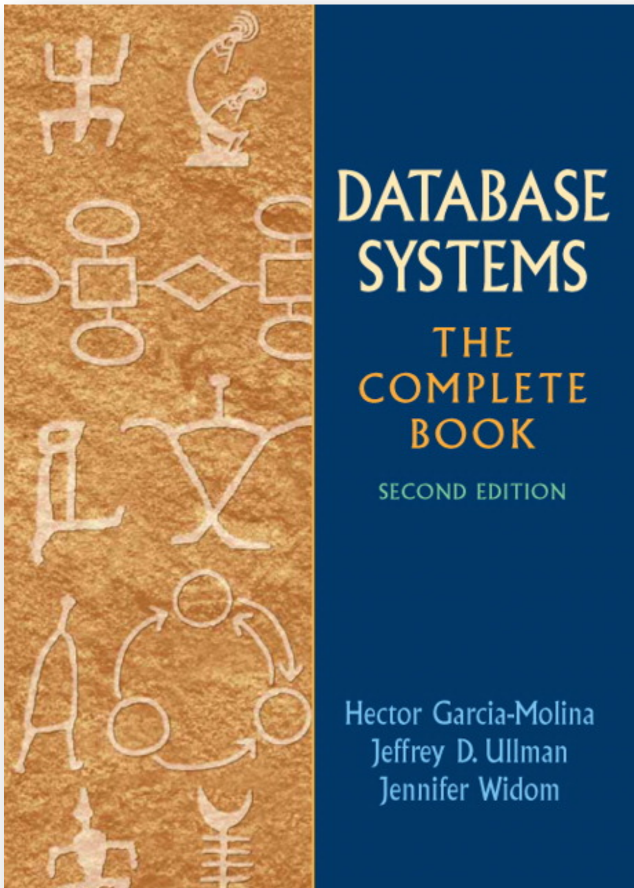
Optional textbook

Assigned paper readings (Q&A in class)

3 programming assignments

2 take-home, open-book tests

Optional Textbook



Database Systems:
The Complete Book

Chapters 13-20

By the original Stanford InfoLab group (Hector Garcia-Molina, Jeff Ullman, Jennifer Widom)

Paper Readings

A few classic or recent research papers

Read the papers **before** class: we want to discuss it together!

We'll post discussion questions on the class website 2-3 weeks before lecture

How Should You Read a Paper?

Read: “How to Read a Paper”

TLDR: don’t just scan end-to-end; focus on key ideas and sections

How to Read a Paper

Version of February 17, 2016

S. Keshav
David R. Cheriton School of Computer Science, University of Waterloo
Waterloo, ON, Canada
keshav@uwaterloo.ca

ABSTRACT

Researchers spend a great deal of time reading research papers. However, this skill is rarely taught, leading to much wasted effort. This article outlines a practical and efficient *three-pass method* for reading research papers. I also describe how to use this method to do a literature survey.

1. INTRODUCTION

Researchers must read papers for several reasons: to review them for a conference or a class, to keep current in their field, or for a literature survey of a new field. A typical researcher will likely spend hundreds of hours every year reading papers.

Learning to efficiently read a paper is a critical but rarely taught skill. Beginning graduate students, therefore, must learn on their own using trial and error. Students waste much effort in the process and are frequently driven to frustration.

For many years I have used a simple ‘three-pass’ approach to prevent me from drowning in the details of a paper before getting a bird’s-eye-view. It allows me to estimate the amount of time required to review a set of papers. Moreover, I can adjust the depth of paper evaluation depending on my needs and how much time I have. This paper describes the approach and its use in doing a literature survey.

2. THE THREE-PASS APPROACH

The key idea is that you should read the paper in up to three passes, instead of starting at the beginning and plowing your way to the end. Each pass accomplishes specific goals and builds upon the previous pass: The *first* pass gives you a general idea about the paper. The *second* pass lets you grasp the paper’s content, but not its details. The *third* pass helps you understand the paper in depth.

4. Read the conclusions
5. Glance over the references, mentally ticking off the ones you’ve already read

At the end of the first pass, you should be able to answer the *five Cs*:

1. *Category*: What type of paper is this? A measurement paper? An analysis of an existing system? A description of a research prototype?
2. *Context*: Which other papers is it related to? Which theoretical bases were used to analyze the problem?
3. *Correctness*: Do the assumptions appear to be valid?
4. *Contributions*: What are the paper’s main contributions?
5. *Clarity*: Is the paper well written?

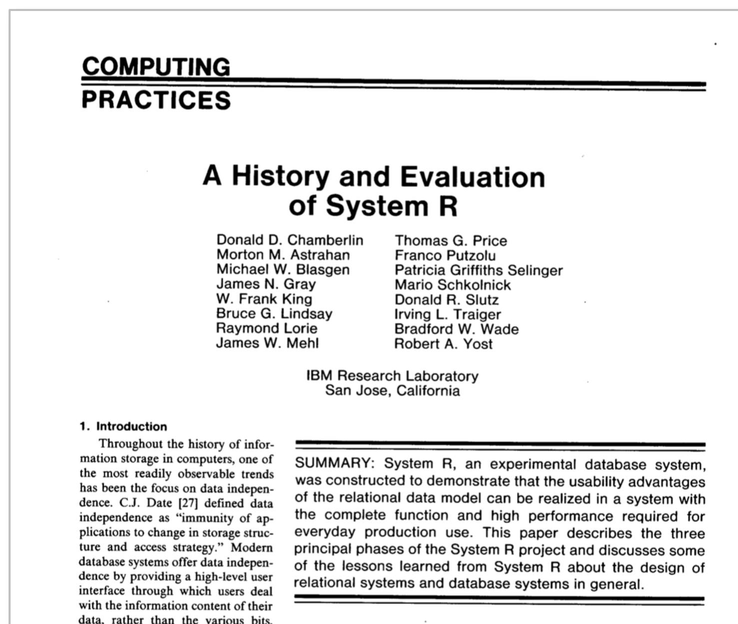
Using this information, you may choose not to read further (and not print it out, thus saving trees). This could be because the paper doesn’t interest you, or you don’t know enough about the area to understand the paper, or that the authors make invalid assumptions. The first pass is adequate for papers that aren’t in your research area, but may someday prove relevant.

Incidentally, when you write a paper, you can expect most reviewers (and readers) to make only one pass over it. Take care to choose coherent section and sub-section titles and to write concise and comprehensive abstracts. If a reviewer cannot understand the gist after one pass, the paper will likely be rejected; if a reader cannot understand the highlights of the paper after five minutes, the paper will likely never be read. For these reasons, a ‘graphical abstract’ that summarizes a paper with a single well-chosen figure is an ex-

Our First Paper

We'll be reading part of "A History and Evaluation of System R" for next class!

Find instructions and questions on website



COMPUTING
PRACTICES

A History and Evaluation of System R

Donald D. Chamberlin
Morton M. Astrahan
Michael W. Blasgen
James N. Gray
W. Frank King
Bruce G. Lindsay
Raymond Lorie
James W. Mehl
Thomas G. Price
Franco Putzolu
Patricia Griffiths Selinger
Mario Schkolnick
Donald R. Slutz
Irving L. Traiger
Bradford W. Wade
Robert A. Yost

IBM Research Laboratory
San Jose, California

SUMMARY: System R, an experimental database system, was constructed to demonstrate that the usability advantages of the relational data model can be realized in a system with the complete function and high performance required for everyday production use. This paper describes the three principal phases of the System R project and discusses some of the lessons learned from System R about the design of relational systems and database systems in general.

1. Introduction

Throughout the history of information storage in computers, one of the most readily observable trends has been the focus on data independence. C.J. Date [27] defined data independence as "immunity of applications to change in storage structure and access strategy." Modern database systems offer data independence by providing a high-level user interface through which users deal with the information content of their data, rather than the various bits,

Programming Assignments

Three assignments implemented in Java or Scala, and submitted online

1. Storage and access methods
2. Query optimization
3. Transactions and recovery

Done individually; A1 posted next week

Midterm and Final

Written tests based on material covered in lectures, assignments and readings

Final will cover the entire course but focus on the second half

Grading

54% Assignments (18% each)

20% Test 1

26% Test 2

Keeping in Touch

Sign up for Piazza on the course website to receive announcements!

cs245.stanford.edu

Outline

Why study data-intensive systems?

Course logistics

Key issues and themes

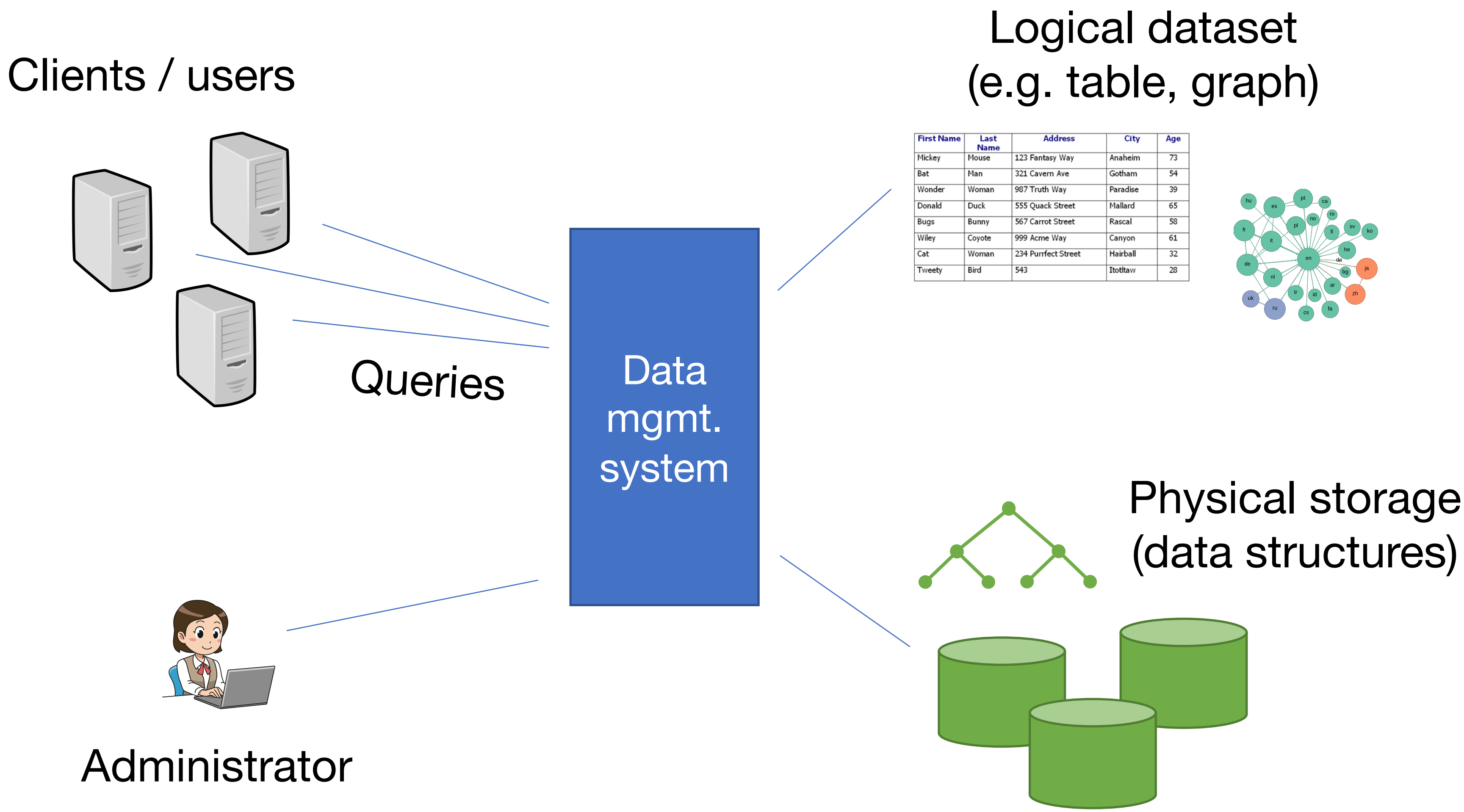
A bit of history

Recall: Examples of Data-Intensive Systems

Relational databases: most popular type of data-intensive system (MySQL, Oracle, etc)

Many systems facing similar concerns: message queues, key-value stores, streaming systems, ML frameworks, your custom app?

Basic Components



First Name	Last Name	Address	City	Age
Mickey	Mouse	123 Fantasy Way	Anaheim	73
Bat	Man	321 Cavern Ave	Gotham	54
Wonder	Woman	987 Truth Way	Paradise	39
Donald	Duck	555 Quack Street	Mallard	65
Bugs	Bunny	567 Carrot Street	Rascal	58
Wiley	Coyote	999 Acme Way	Canyon	61
Cat	Woman	234 Purrfect Street	Hairball	32
Tweety	Bird	543	Itotltaw	28

Examples

System	Logical Data Model	Physical Storage	API	Other Features
Relational databases	Relations (i.e. tables)	B-trees, column stores, indexes, …	SQL, ODBC	Durability, transactions, query planning, migrations, …

Examples

System	Logical Data Model	Physical Storage	API	Other Features
Relational databases	Relations (i.e. tables)	B-trees, column stores, indexes, …	SQL, ODBC	Durability, transactions, query planning, migrations, …
TensorFlow				

Examples

System	Logical Data Model	Physical Storage	API	Other Features
Relational databases	Relations (i.e. tables)	B-trees, column stores, indexes, …	SQL, ODBC	Durability, transactions, query planning, migrations, …
TensorFlow	Tensors			

Examples

System	Logical Data Model	Physical Storage	API	Other Features
Relational databases	Relations (i.e. tables)	B-trees, column stores, indexes, …	SQL, ODBC	Durability, transactions, query planning, migrations, …
TensorFlow	Tensors	NCHW, NHWC, sparse arrays, …		

Examples

System	Logical Data Model	Physical Storage	API	Other Features
Relational databases	Relations (i.e. tables)	B-trees, column stores, indexes, …	SQL, ODBC	Durability, transactions, query planning, migrations, …
TensorFlow	Tensors	NCHW, NHWC, sparse arrays, …	Python DAG construction	

Examples

System	Logical Data Model	Physical Storage	API	Other Features
Relational databases	Relations (i.e. tables)	B-trees, column stores, indexes, …	SQL, ODBC	Durability, transactions, query planning, migrations, …
TensorFlow	Tensors	NCHW, NHWC, sparse arrays, …	Python DAG construction	query planning, distribution, specialized HW

Examples

System	Logical Data Model	Physical Storage	API	Other Features
Relational databases	Relations (i.e. tables)	B-trees, column stores, indexes, …	SQL, ODBC	Durability, transactions, query planning, migrations, …
TensorFlow	Tensors	NCHW, NHWC, sparse arrays, …	Python DAG construction	query planning, distribution, specialized HW
Apache Kafka				

Examples

System	Logical Data Model	Physical Storage	API	Other Features
Relational databases	Relations (i.e. tables)	B-trees, column stores, indexes, …	SQL, ODBC	Durability, transactions, query planning, migrations, …
TensorFlow	Tensors	NCHW, NHWC, sparse arrays, …	Python DAG construction	query planning, distribution, specialized HW
Apache Kafka	Streams of opaque records	Partitions, compaction	Publish, subscribe	Durability, rescaling

Examples

System	Logical Data Model	Physical Storage	API	Other Features
Relational databases	Relations (i.e. tables)	B-trees, column stores, indexes, …	SQL, ODBC	Durability, transactions, query planning, migrations, …
TensorFlow	Tensors	NCHW, NHWC, sparse arrays, …	Python DAG construction	query planning, distribution, specialized HW
Apache Kafka	Streams of opaque records	Partitions, compaction	Publish, subscribe	Durability, rescaling
Apache Spark RDDs				

Examples

System	Logical Data Model	Physical Storage	API	Other Features
Relational databases	Relations (i.e. tables)	B-trees, column stores, indexes, …	SQL, ODBC	Durability, transactions, query planning, migrations, …
TensorFlow	Tensors	NCHW, NHWC, sparse arrays, …	Python DAG construction	query planning, distribution, specialized HW
Apache Kafka	Streams of opaque records	Partitions, compaction	Publish, subscribe	Durability, rescaling
Apache Spark RDDs	Collections of Java objects	Read external systems, cache	Functional API, SQL	Distribution, query planning, transactions*

Some Typical Concerns

Access interface from many, changing apps

Performance: throughput, latency, etc

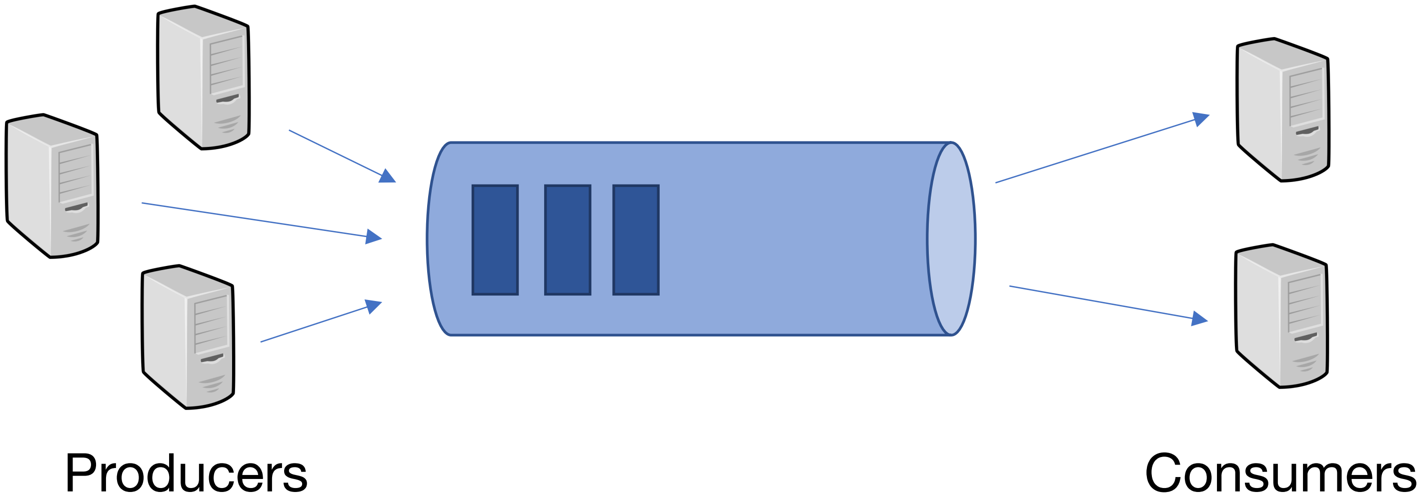
Reliability in the face of hardware crashes, bugs, bad user input, etc

Concurrency: access by multiple users

Security and data privacy

Example

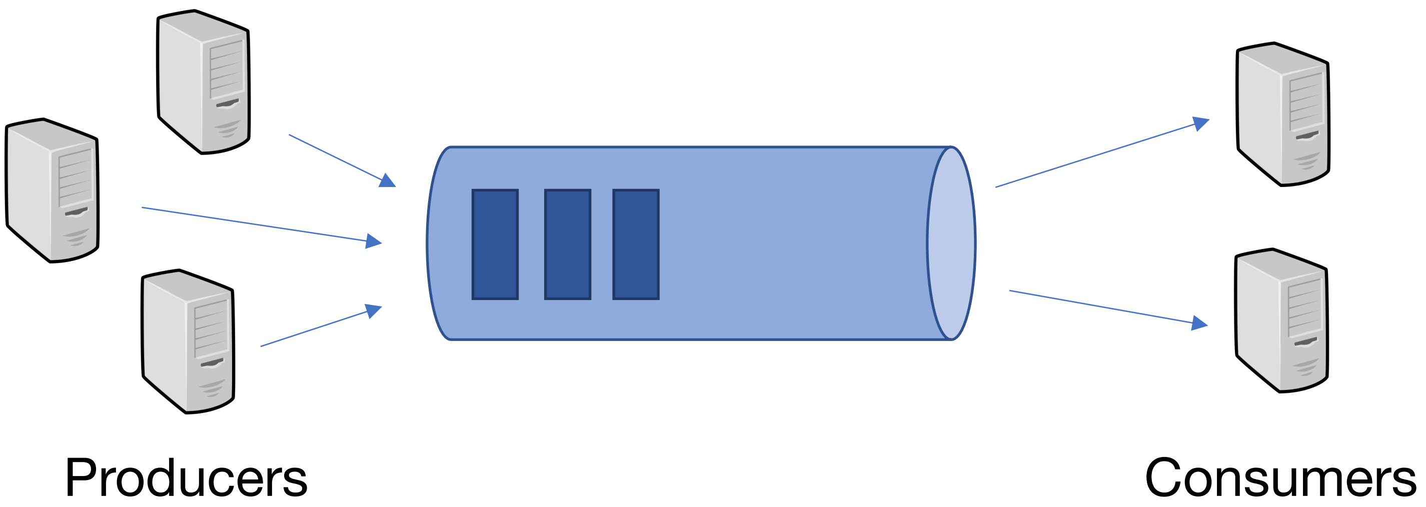
Message queue system



What should happen if two consumers read() at the same time?

Example

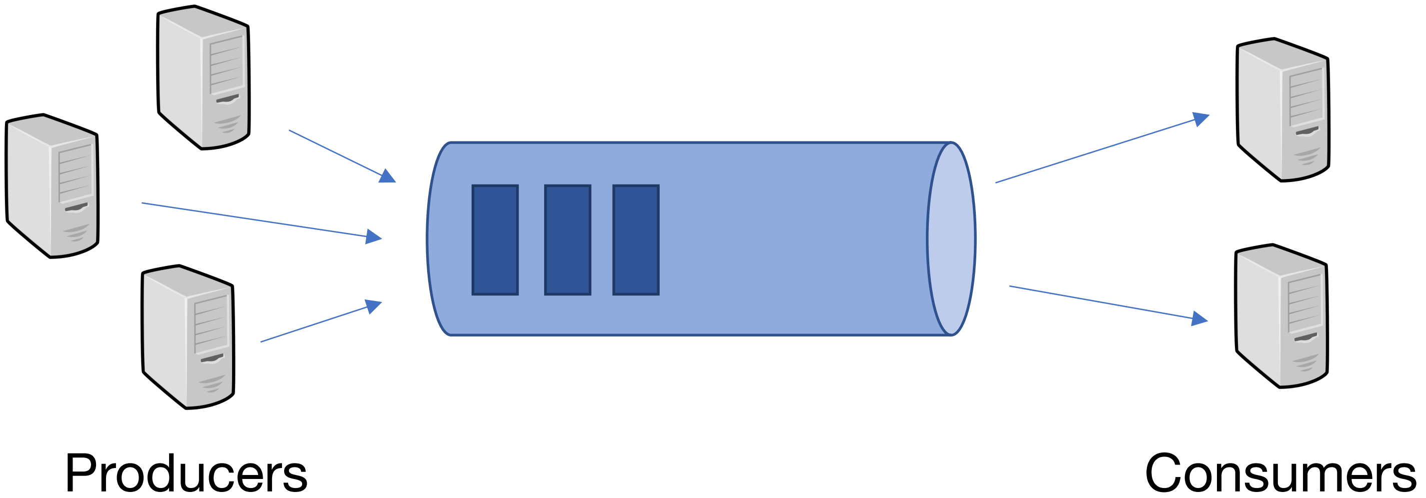
Message queue system



What should happen if a consumer reads a message but then immediately crashes?

Example

Message queue system



Can a producer put in 2 messages atomically?

Two Big Ideas

Declarative interfaces

» Apps specify *what* they want, not *how* to do it

» Example: “store a table with 2 integer columns”, but not how to encode it on disk

» Example: “count records where column1 = 5”

Transactions

» Encapsulate multiple app actions into one *atomic* request (fails or succeeds as a whole)

» Concurrency models for multiple users

» Clear interactions with failure recovery

Declarative Interface Examples

SQL

- » Abstract “table” data model, many physical implementations
- » Specify queries in a restricted language that the database can optimize

TensorFlow

- » Operator graph gets mapped & optimized to different hardware devices

Functional programming (e.g. MapReduce)

- » Says what to run but not how to do scheduling

Transaction Examples

SQL databases

» Commands to start, abort or end transactions based on multiple SQL statements

Apache Spark, MapReduce

» Make the multi-part output of a job appear atomically when all partitions are done

Stream processing systems

» Count each input record exactly once despite crashes, network failures, etc

Outline

Why study data-intensive systems?

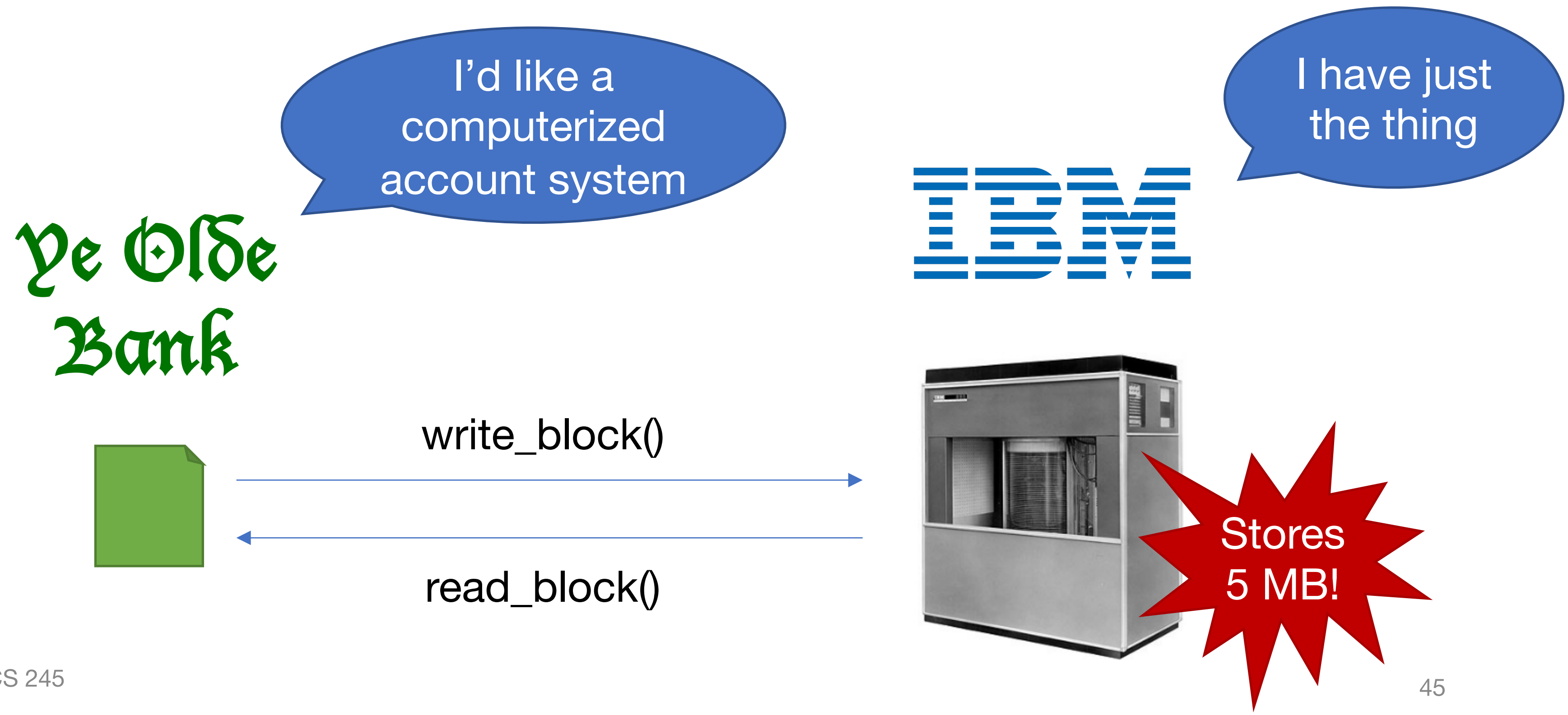
Course logistics

Key issues and themes

A bit of history

Early Data Management

At first, each application did its own data management directly against storage



Problems with App Storage Management

How should we lay out and navigate data?

How do we keep the application reliable?

What if we want to share data across apps?

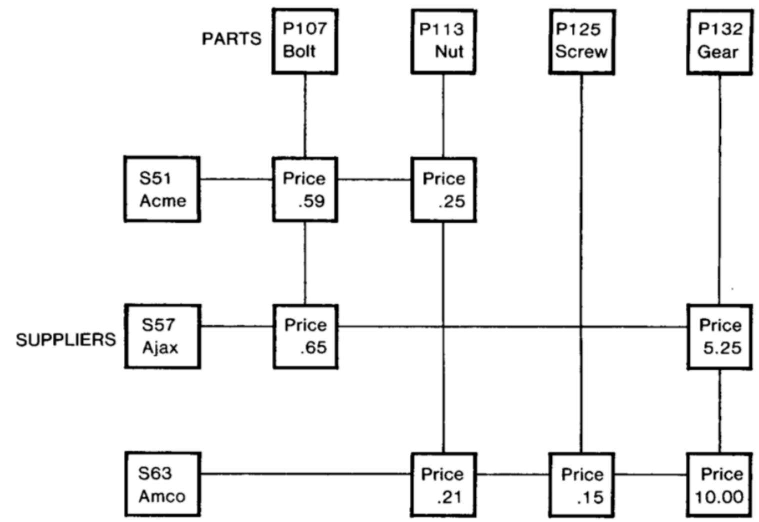
Every app is solving the *same* problems!

Navigational Databases (1964)

CODASYL, IDS

Data is graph of records

Procedural API based on navigating links:




```
get department with name='Sales'
get first employee in set department-employees
until end-of-set do {
  get next employee in set department-employees
  process employee
}
```

“Data independence”: app code is not tied to storage details

I raise the example of Copernicus today to illustrate a parallel that I believe exists in the computing or, more properly, the information systems world. We have spent the last 50 years with almost Ptolemaic information systems. These systems, and most of the thinking about systems, were based on a "computer centered" concept.

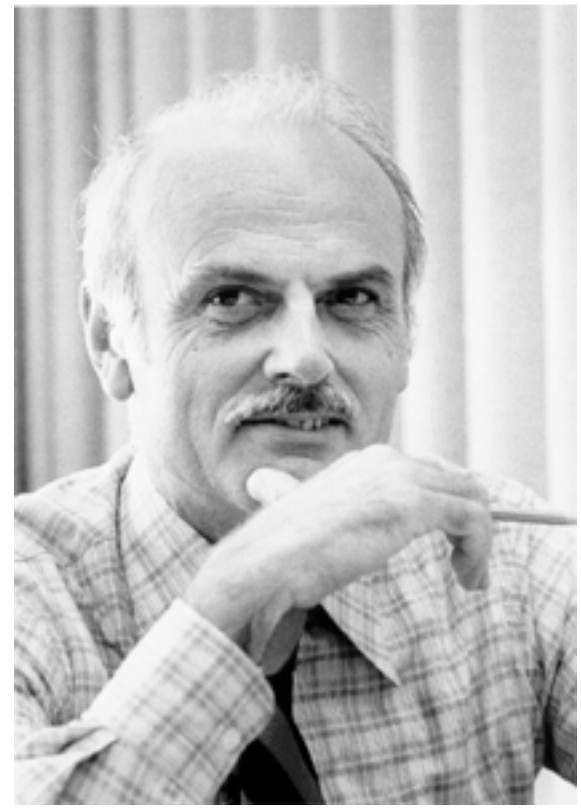
A new basis for understanding is available in the area of information systems. It is achieved by a shift from a computer-centered to the database-centered point of view. This new understanding will lead to new solutions to our database problems and speed our conquest of the *n*-dimensional data structures which best model the complexities of the real world.

Charles W. Bachman, "The Programmer as Navigator"

Edgar F. (Ted) Codd

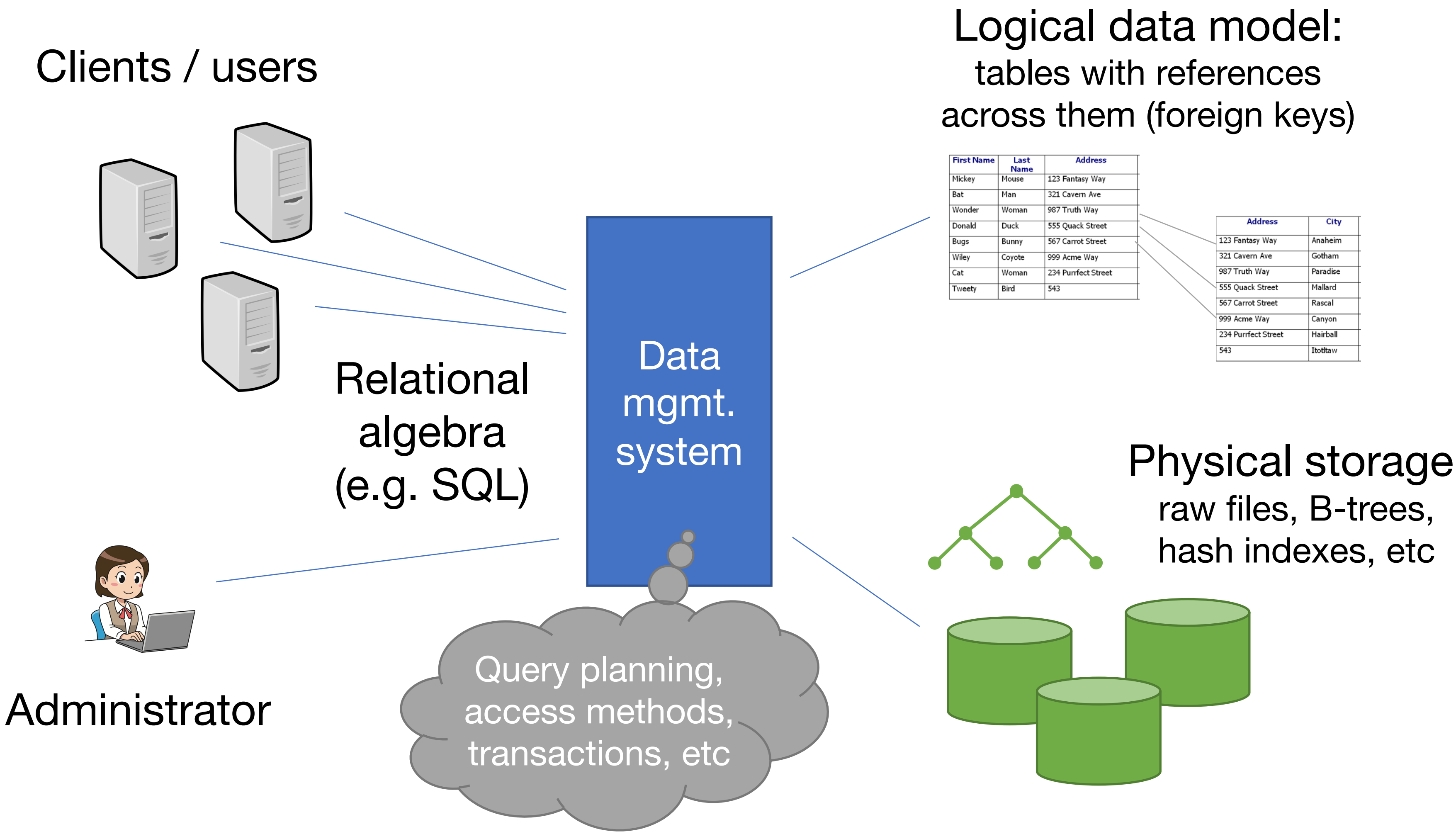
Proposed the **relational** DB model, with declarative queries & storage (1970)

Relation = table with unique key identifying each row



Data independence++: apps don't even specify how to execute queries

Key Ideas in Relational DBMS



First Name	Last Name	Address
Mickey	Mouse	123 Fantasy Way
Bat	Man	321 Cavern Ave
Wonder	Woman	987 Truth Way
Donald	Duck	555 Quack Street
Bugs	Bunny	567 Carrot Street
Wiley	Coyote	999 Acme Way
Cat	Woman	234 Purrfect Street
Tweety	Bird	543

Address	City
123 Fantasy Way	Anaheim
321 Cavern Ave	Gotham
987 Truth Way	Paradise
555 Quack Street	Mallard
567 Carrot Street	Rascal
999 Acme Way	Canyon
234 Purrfect Street	Hairball
543	Itotltaw

Early Relational DBMS

IBM System R (1974): research system

» Led to IBM SQL/DS in 1981

Ingres (1974): Mike Stonebraker at Berkeley

» Led to PostgreSQL

Oracle database (released 1979)

Next class, we'll cover database architecture by looking at System R

Rest of the Course

We'll explore both "big ideas" we saw, focusing on relational DBs but showing examples in other areas

- Declarative interfaces
  - Data independence and data storage formats
  - Query languages and optimization
- Transactions, concurrency & recovery
  - Concurrency models
  - Failure recovery
  - Distributed storage and consistency

Don't forget to sign up for Piazza!