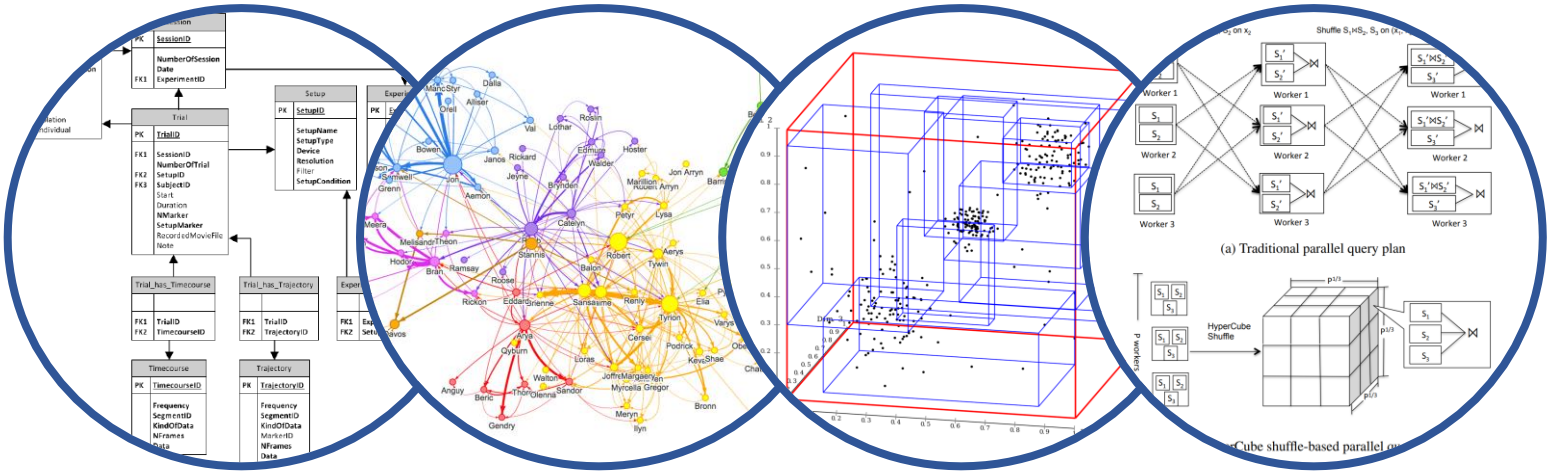



# Introduction to Data Management

## Databases in Theory and Practice

Shana Hutchison
Adapted from Jonathan Leang, Ryan Maas, Alyssa Pittman, …

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

# First, a story…

The Seattle Times

Education | Local News

## New UW payroll system behind schedule, more costly than expected

Originally published November 26, 2015 at 2:36 pm | *Updated November 27, 2015 at 6:19 am*

**A project to modernize the University of Washington's payroll system is costing millions more and taking longer than expected.**

# Outline

1. Administrivia
2. The Relational Data Model
3. Databases, SQL, and RA

# What am I going to learn?

- Course Topics
  - Data Models*
  - Queries
  - Database Design
  - Optimization
  - Transactions and Parallelism
  - Semi-Structured NoSQL Databases
- Tools:
  - Portable through Enterprise platforms
  - Cloud Services (AWS, Microsoft Azure)

# What am I going to learn?

- After the course, you will be able to…
  - Explain how a query is processed end-to-end
  - Integrate a database into an application
  - Effectively manage data for long-term use
  - Optimize a database for speed or storage
  - Make design choices when selecting tools for a project
    - "No one size fits all!"

— *Michael Stonebraker*, Turing Awardee

# 414 Staff

Instructor: Shana Hutchison (shutchis @ cs)

- She/her pronouns

## Teaching Assistants

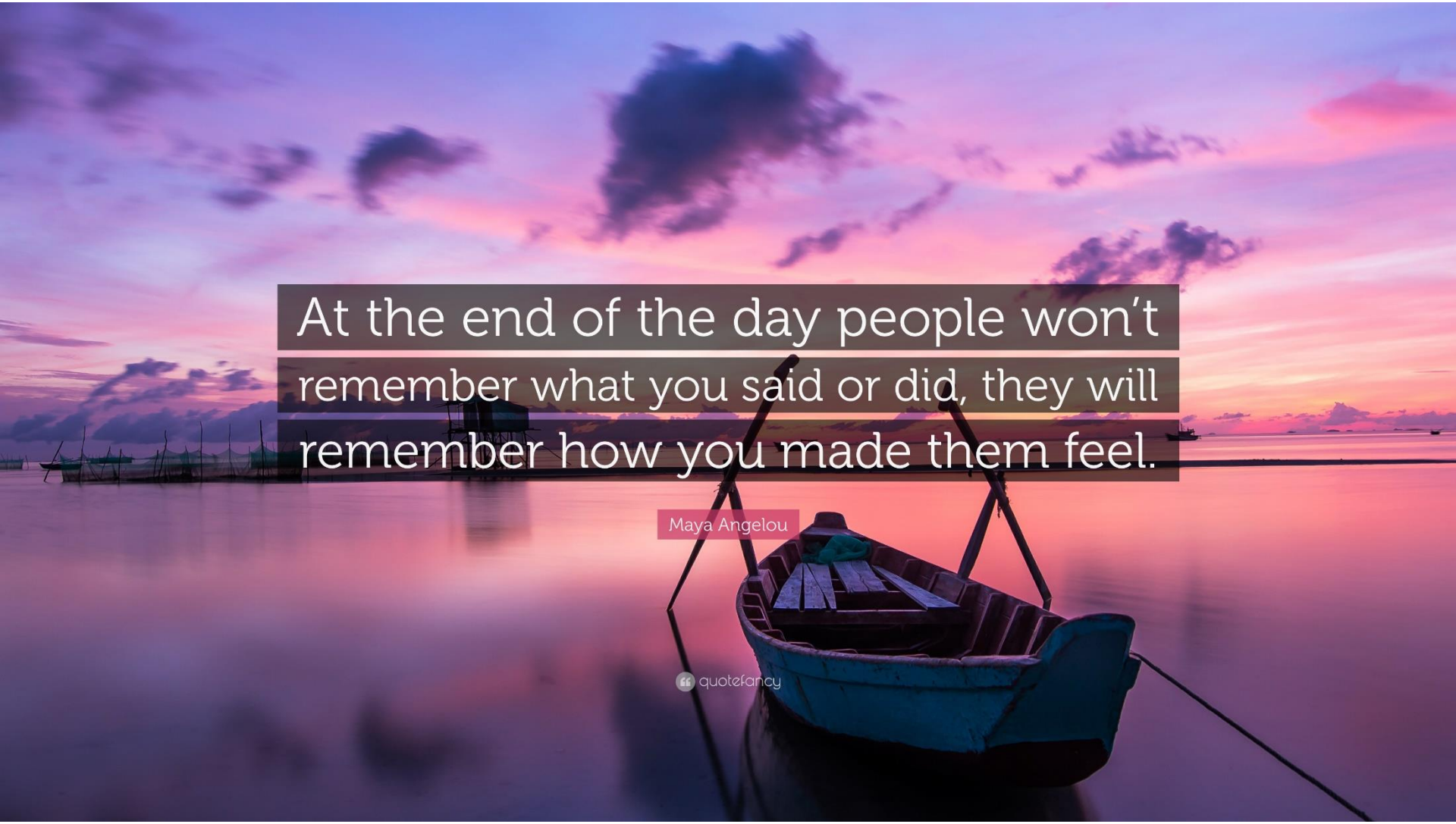
- Wenjun Chen
- Xinyue Chen
- Gibbs Geng
- Ryan Huang
- Steve Ma
- Pranay Mundra
- Steven Su
- McKinnon Frei Williams
- Cong Yan
- Jack Zhang

See website / calendar for office hours!

# Course Format

- Lectures: this room, please attend!
  - Panopto recorded for UW student / staff use
  - See me for privacy concerns
- Sections: for locations, see website
  - Bring your laptop, or sit next to a friend who has one
- Midterm & Final (in-class)
- Participate in Piazza discussion
- 7 homework assignments
  - hw1: practice partner
  - hw2, hw3: section partner #1
  - hw4, hw5: section partner #2
  - hw6, hw7: section partner #3

# Collaboration



# Collaboration

- Teamwork & empathy
  - Just like the real world! Soft skills matter
  - Teams that meet in person generally score higher
    - Schedule time. See your partner @ lecture & section.
  - Acknowledge power differences
    - Background knowledge, age, income, race, gender, …
  - Feel lost? Ask your partner
    Feel like you're doing all the work? Teach your partner
    - Teaching is the highest form of learning. Trust me ( ͡° ͜ʖ ͡°)
- Every team must write their own solution
  - But do discuss course topics!

Ref: https://faculty.washington.edu/ajko/books/cooperative-software-development/

# Collaboration

- Strategies
  1. In-person *pair programming*: one typing, one navigating
     - Try this on hw1! Switch roles after a while
  2. Use online tools
     - Google Hangout, Skype, collabedit.com, …
  3. Each do the assignment individually, then merge solns.
     - Leave time to compare and discuss
     - Git branches may help
- Best not to split up an assignment
  - You only learn / practice half the material
- Did one "do all the work"?
  - First: talk about it. If unresolved, see TA & leave feedback

Ref: https://web.stanford.edu/class/archive/cs/cs106a/cs106a.1178/assignments/pair.html

# Feedback

- Mandatory. Please be honest & constructive
- Grading based on feedback over 3 partnerships

- Homework grading: *Correctness*
- Team dynamic grading:
  - *Contribution*
  - *Communication*
  - *Reflection*

- First hw feedback is practice / ungraded

# Team Reflection Qs

Shared with partner:

- What are you most grateful to have experienced/learned?
- Rate & explain the **contribution** of you & your partner
- Rate & explain the **communication** of you & your partner

Confidential (visible only to 414 staff):

- Comments on your team's dynamic
- What would you do differently in future groupwork?
- Would you like to meet in-person to discuss further?

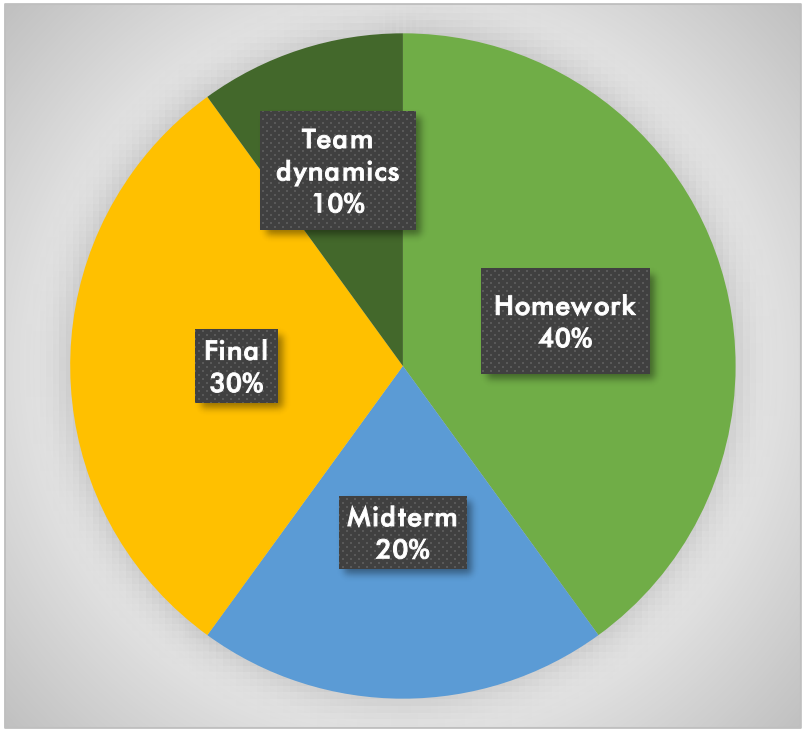
# Homework late policy

- One free late day per partnership
  - A "late day" is a 24-hour period
  - Does not carry over across partnerships
- Additional late days at 10% penalty
- Hard deadline: no submission accepted after two days after the due date (max 2 late days)

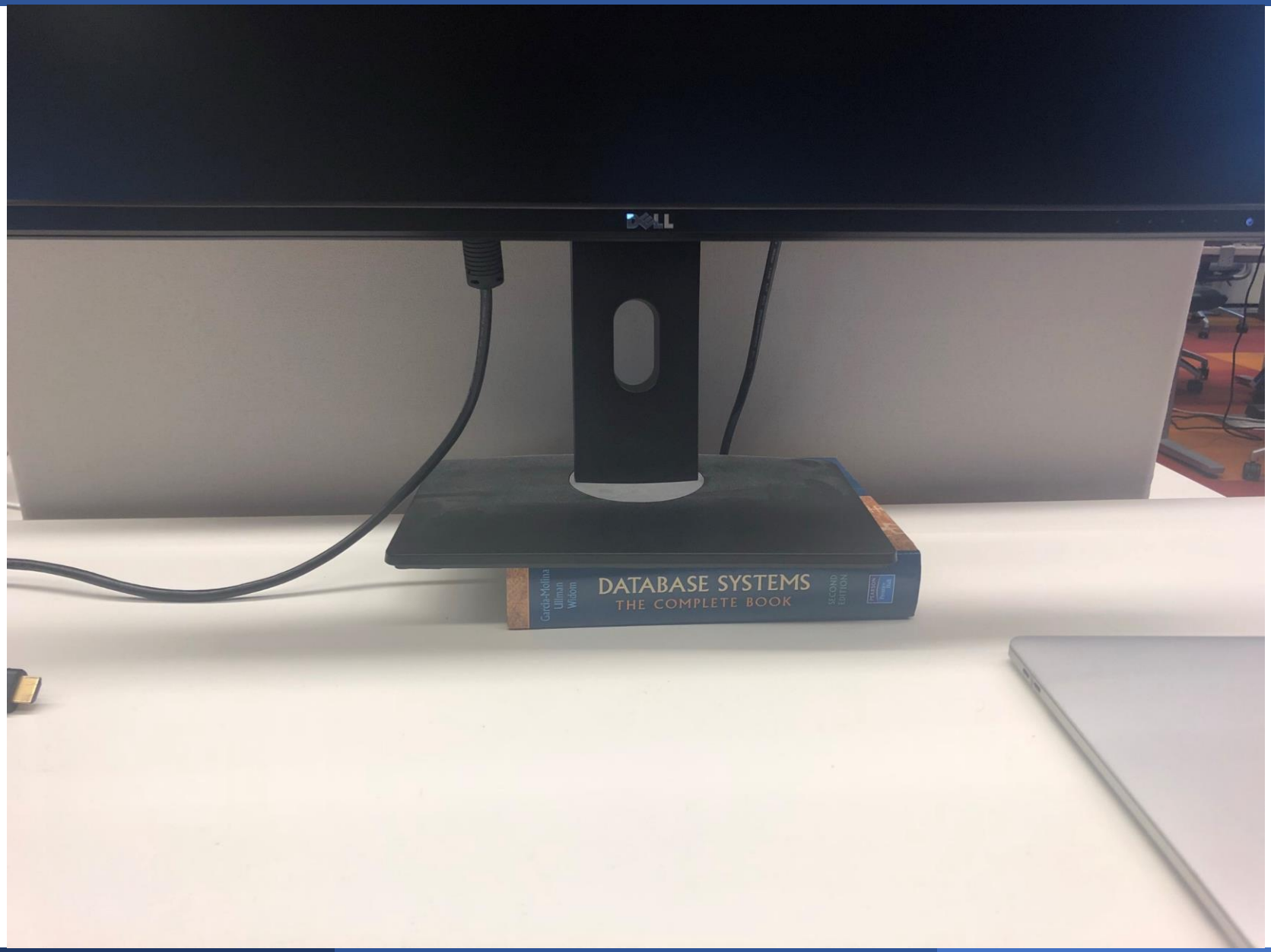
# Exams

- Midterm (TBD) and Final (March 19)
- Allowed note sheet
  - **Handwritten**
  - Letter-size
  - May write on both sides
  - Midterm: 1 sheet, Final: 2 sheets
- Closed book. No computers, phones, watches, …
- Past exams with solutions linked on website

# Grading



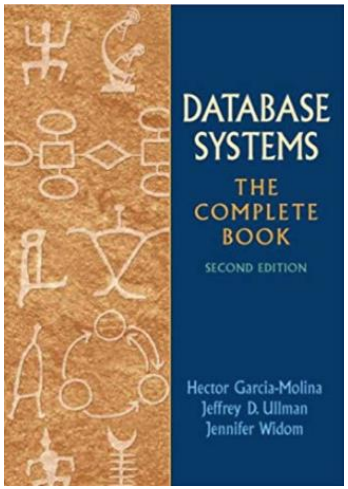
# Textbook


DELL
Garcia-Molina
Ullman
Widom
DATABASE SYSTEMS
THE COMPLETE BOOK
SECOND EDITION

# Textbook

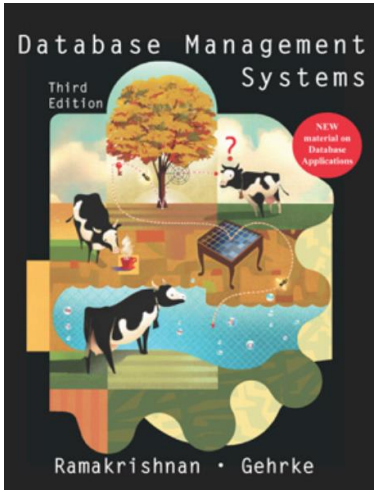
Main textbook, available at the bookstore or pdf:

- *Database Systems: The Complete Book*,
  Hector Garcia-Molina,
  Jeffrey Ullman,
  Jennifer Widom,
  *second edition*.




Also useful:

- *Database Management Systems*
  (3rd Edition)



# Communication

- Web page: http://www.cs.washington.edu/414
- Piazza message board https://piazza.com/class/k46r2oee1lb2vk
  - **THE** place to ask course-related questions
  - Log in today, enable notifications
  - Opt-out of the Piazza Network, unless you want to disclose your activity to companies
  - Please minimize anonymous posts
    - Asking for help is a sign of strength!
- Class mailing list
  - Occasional announcements
- UW Canvas
  - Grades & Panopto lecture recordings

# Recap on Resources

- Panopto
- Office Hours
- Piazza
- Textbooks
- Software documentation
  - Often better than Google!
- Your fellow students, especially partners

# Let’s get started!

# Database

What is a database ?

Give examples of databases

# Database

What is a database ?

- A collection of related data

Give examples of databases

# Database

What is a database ?

- A collection of related data

Give examples of databases

- Payroll
- UW students
- Amazon's products
- Airline reservations
- Dating app profiles
- Web traffic
- Your notes for this class

# Database Management System

## What is a DBMS ?

- *A program to efficiently manage and persist a database over long periods of time*

## Examples of DBMSs

- Oracle, IBM DB2, Microsoft SQL Server, Vertica, Teradata
- Open source: MySQL (Sun/Oracle), PostgreSQL, CouchDB
- Open source library: SQLite

## We will focus on relational DBMSs most quarter

# DBMS Properties

- Queryable. Ask your DB questions
- Durable (data persists power-off). Or not!
  - Ex. In-memory DB trade durability for speed
- Have *schema*. Or not!
  - Ex. Semi-structured DB
- No redundancy. Or not!
  - *Indexes* trade space for speed
- Optimizes your queries. Or not!
  - NoSQL DB has a "WYSIWYG" flavor
- Correctly handles concurrent access. Or not!
  - *Serializability* can be turned off for speed

# Data Management is all about tradeoffs!

# DBMS Data

**Data Model**

A **Data Model** is a mathematical formalism to describe data. It is how we can talk about data conceptually without having to think about implementation.

# 3 Parts of a Data Model

- Instance
  - The actual **data**
- Schema
  - A **description** of what data is being stored
- Query Language
  - How to retrieve and manipulate data

# Data Model Zoo

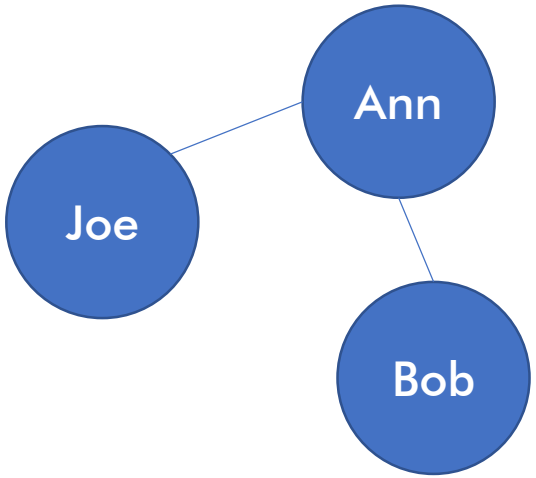
There are lots of models out there!

- Relational
- Semi-structured
- Key-value
- Graph
- Matrix
- Object-oriented
- RDF (subject-predicate-object)
- …

# Multiple Representation

Data can be represented in different ways

An example of Facebook friends




| Person1 | Person2 | Friend |
|---|---|---|
| Joe | Ann | 1 |
| Ann | Bob | 1 |
| Bob | Joe | 0 |

$$\begin{array}{c} \\ Joe \\ Ann \\ Bob \end{array} \begin{array}{c} \begin{array}{ccc} Joe & Ann & Bob \end{array} \\ \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \end{array}$$

**Graph model** **Relational Model** **Matrix Model**

# What is the Relational Model?



## A Relational Model of Data for Large Shared Data Banks

E. F. CODD
*IBM Research Laboratory, San Jose, California*

Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users at terminals and most application programs should remain

The relational view (or model) of data described in Section 1 appears to be superior in several respects to the graph or network model [3, 4] presently in vogue for non-inferential systems. It provides a means of describing data with its natural structure only—that is, without superimposing any additional structure for machine representation purposes. Accordingly, it provides a basis for a high level data language which will yield maximal independence between programs on the one hand and machine representation and organization of data on the other.

A further advantage of the relational view is that it forms a sound basis for treating derivability, redundancy, and consistency of relations—these are discussed in Section

systems has been to deductive question-answering systems. Levein and Maron [2] provide numerous references to work in this area.

In contrast, the problems treated here are those of *data independence*—the independence of application programs and terminal activities from growth in data types and changes in data representation—and certain kinds of *data inconsistency* which are expected to become troublesome even in nondeductive systems.

element to participate in several orderings. Let us consider those existing systems which either require or permit data elements to be stored in at least one total ordering which is closely associated with the hardware-determined ordering of addresses. For example, the records of a file concerning parts might be stored in ascending order by part serial number. Such systems normally permit application programs to assume that the order of presentation of records from such a file is identical to (or is a subordering of) the

# What is the Relational Model?



## A Relational Model of Data for Large Shared Data Banks

E. F. Codd
*IBM Research Laboratory, San Jose, California*

Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users at terminals and most application programs should

The relational view (or model) of data described in Section 1 appears to be superior in several respects to the graph or network model [3, 4] inferential systems. It provides with its natural structure only posing any additional structure purposes. Accordingly, it provi data language which will yield tween programs on the one ha tion and organization of data

A further advantage of the forms a sound basis for treatin and consistency of relations—t


systems has been to deductive question-answering Levein and Maron [2] provide numerous reference in this area.

In contrast, the problems treated here are tho *independence*—the independence of application and terminal activities from growth in data t changes in data representation—and certain kin *inconsistency* which are expected to become tr even in nondeductive systems.

Volume 13 / Number 6 / June, 1970

| Rank Jan 2019 | Rank Dec 2018 | Rank Jan 2018 | DBMS | Database Model |
|---|---|---|---|---|
| 1. | 1. | 1. | Oracle | Relational DBMS |
| 2. | 2. | 2. | MySQL | Relational DBMS |
| 3. | 3. | 3. | Microsoft SQL Server | Relational DBMS |
| 4. | 4. | 4. | PostgreSQL | Relational DBMS |
| 5. | 5. | 5. | MongoDB | Document store |
| 6. | 6. | 6. | IBM Db2 | Relational DBMS |
| 7. | 7. | ↑ 9. | Redis | Key-value store |
| 8. | 8. | ↑ 10. | Elasticsearch | Search engine |
| 9. | 9. | ↓ 7. | Microsoft Access | Relational DBMS |
| 10. | 10. | ↑ 11. | SQLite | Relational DBMS |

https://db-engines.com/en/ranking

# Components of the Relational Model

Payroll (UserId, Name, Job, Salary)

# Components of the Relational Model

Payroll (UserId, Name, Job, Salary)

# Components of the Relational Model

**Payroll**

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000* |
| 345 | Allison | TA | 60000* |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

* I wish

# Components of the Relational Model

**Payroll**

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000* |
| 345 | Allison | TA | 60000* |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

Instance of actual data

* I wish

# Components of the Relational Model

**Table/ Relation**

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

# Components of the Relational Model

**Table/ Relation**

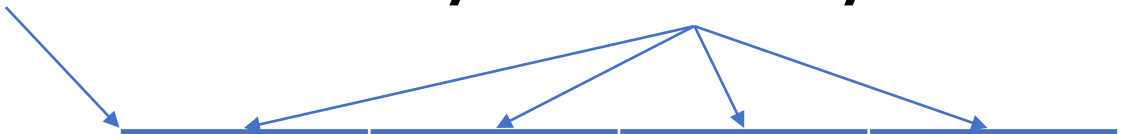
**Rows/ Tuples/ Records**

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

# Components of the Relational Model

**Table/ Relation**

**Columns/Attributes/Fields**



| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

**Rows/ Tuples/ Records**

# Characteristics of the Relational Model

- Originally defined with **Set semantics**
  - No duplicate tuples
- Attributes are **typed** and **static**
  - INTEGER, FLOAT, VARCHAR(n), DATETIME, …
- Tables are **flat**

# Characteristics of the Relational Model

- Originally defined with **Set semantics**
  - No duplicate tuples
- Attributes are **typed** and **static**
  - INTEGER, FLOAT, VARCHAR(n), DATETIME, …
- Tables are **flat**

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

# Characteristics of the Relational Model

- Originally defined with **Set semantics**
  - No duplicate tuples
- Attributes are **typed** and **static**
  - INTEGER, FLOAT, VARCHAR(n), DATETIME, …
- Tables are **flat**

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

{ (123, Jack, TA, 50000),
(345, Allison, TA, 60000),
(567, Magda, Prof, 90000),
(789, Dan, Prof, 100000) }

Tables are fancy ways to display sets

# Characteristics of the Relational Model

- Originally defined with **Set semantics**
  - No duplicate tuples
- Attributes are **typed** and **static**
  - INTEGER, FLOAT, VARCHAR(n), DATETIME, …
- Tables are **flat**

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

=

| UserID | Name | Job | Salary |
|---|---|---|---|
| 567 | Magda | Prof | 90000 |
| 123 | Jack | TA | 50000 |
| 789 | Dan | Prof | 100000 |
| 345 | Allison | TA | 60000 |

Order doesn't matter

# Characteristics of the Relational Model

- Originally defined with **Set semantics**
  - No duplicate tuples
- Attributes are **typed** and **static**
  - INTEGER, FLOAT, VARCHAR(n), DATETIME, …
- Tables are **flat**

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |
| 789 | Dan | Prof | 100000 |

Violates set semantics!

# Characteristics of the Relational Model

- **Set semantics** → not in most DBMS implementations
  - No duplicate tuples
- Attributes are **typed** and **static**
  - INTEGER, FLOAT, VARCHAR(n), DATETIME, …
- Tables are **flat**

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |
| 789 | Dan | Prof | 100000 |

Violates set semantics!

# Characteristics of the Relational Model

- **Set semantics** → not in most DBMS implementations
  - No duplicate tuples
- Attributes are **typed** and **static**
  - INTEGER, FLOAT, VARCHAR(n), DATETIME, …
- Tables are **flat**

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | banana |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

Violates attribute type assuming INT

# Characteristics of the Relational Model

- **Set semantics** → not in most DBMS implementations
  - No duplicate tuples
- Attributes are **typed** and **static**
  - INTEGER, FLOAT, VARCHAR(n), DATETIME, …
- Tables are **flat**

No sub-tables allowed!

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | JobName: TA, HasBananas: 0<br>JobName: banana picker, HasBananas: 1 | 0000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

# Characteristics of the Relational Model

But how is this data ACTUALLY stored?

**Payroll**

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

# Characteristics of the Relational Model

But how is this data ACTUALLY stored?

**Payroll**

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

Don't know. Don't care.

**Physical Data Independence**

# Structured Query Language - SQL

Alright, I have data and a schema.

How do I access it?

# Structured Query Language - SQL

"**SQL (standing for Structured Query Language) is the standard language for relational database management systems.** When it originated back in the 1970s, the **domain-specific language** was intended to fulfill the need of conducting a database query that could navigate through a network of pointers to find the desired location. Its application in handling structured data has fostered in the Digital Age. In fact, the powerful database manipulation and definition capabilities of SQL and its intuitive tabular view have become available in some form on virtually every important computer platform in the world.

**Some notable features of SQL include the ability to process sets of data as groups instead of individual units, automatic navigation to data, and the use of statements that are complex and powerful individually.** Used for a variety of tasks, such as querying data, controlling access to the database and its objects, guaranteeing database consistency, updating rows in a table, and creating, replacing, altering and dropping objects, **SQL lets users work with data at the logical level.**"

Read more at the ANSI Blog: The SQL Standard – ISO/IEC 9075:2016 https://blog.ansi.org/?p=158690

# Structured Query Language - SQL

- Key points about SQL:
  - A domain-specific language
    - SQL only works on relational databases
    - Not for general purpose programming (Java, C/C++, …)
  - **Logical level of interaction with data**

# Hello World

**Payroll**

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

```
SELECT P.Name, P.UserID
  FROM Payroll AS P
 WHERE P.Job = 'TA';
```

# Hello World

**Payroll**

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

```
SELECT P.Name, P.UserID
  FROM Payroll AS P
 WHERE P.Job = 'TA';
```

**FROM**
Where the data coming from

# Hello World

**Payroll**

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

```
SELECT P.Name, P.UserID
  FROM Payroll AS P
 WHERE P.Job = 'TA';
```

**FROM**
Where the data coming from

**WHERE**
Filter the data

# Hello World

**Payroll**

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

```
SELECT  P.Name, P.UserID
  FROM  Payroll AS P
 WHERE  P.Job = 'TA';
```

**SELECT**
What kind of data I want

**FROM**
Where the data coming from

**WHERE**
Filter the data

# Hello World

**Payroll**

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

| Name | UserID |
|---|---|
| Jack | 123 |
| Allison | 345 |

```
SELECT  P.Name, P.UserID
  FROM  Payroll AS P
 WHERE  P.Job = 'TA';
```

# Think About This

Wait!

How does a computer understand abstract SQL text?

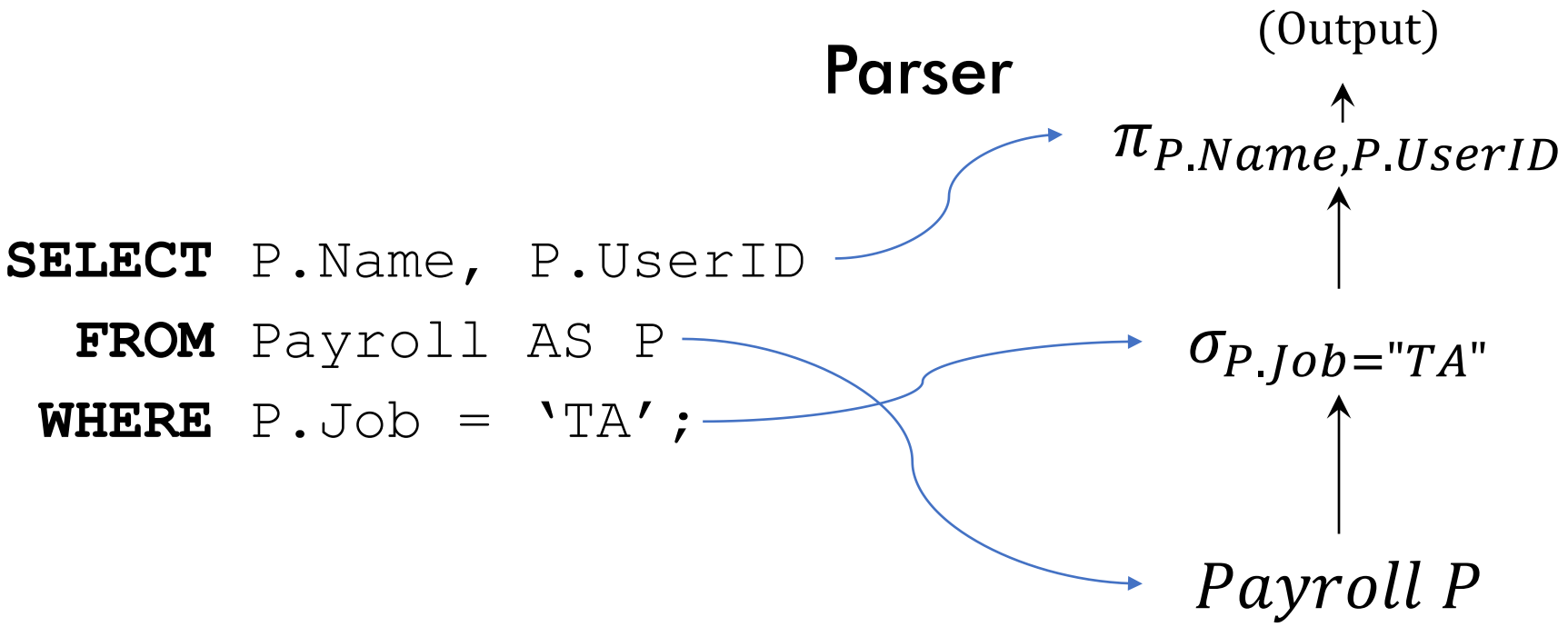
# Database Internals

- How code boils down to instructions
- Relational Database Management Systems (RDBMSs) use **Relational Algebra** (RA)

```
SELECT P.Name, P.UserID
  FROM Payroll AS P
 WHERE P.Job = 'TA';
```

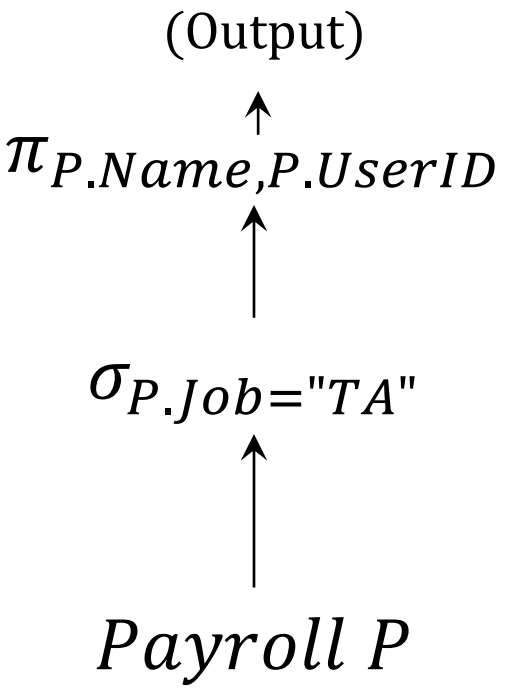
# SQL to RA

- How code boils down to instructions
- Relational Database Management Systems (RDBMSs) use **Relational Algebra** (RA).

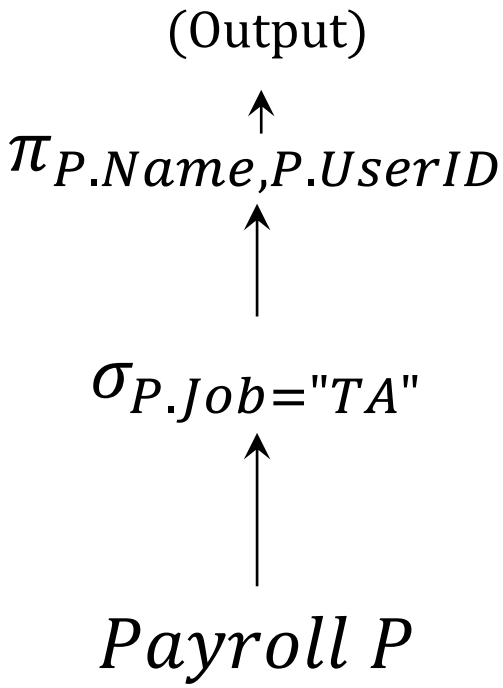


# RA is Math

- Relational Algebra is math!
  - Operators on sets
  - Defines how to compute a query
  - Operators covered later
- For now, let's explore one way to execute an RA plan



# RA: Iterator / "for-each" semantics

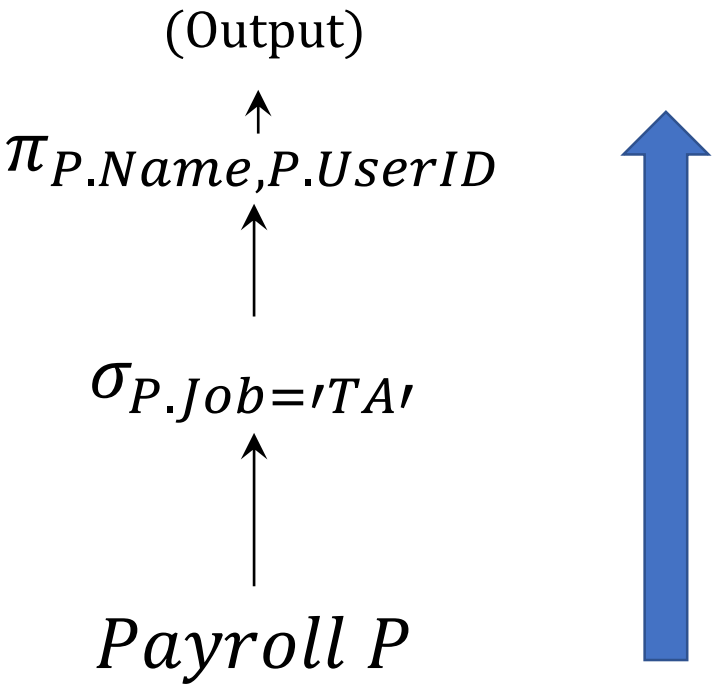



```
SELECT P.Name, P.UserID
  FROM Payroll AS P
 WHERE P.Job = 'TA';
```

**For-each semantics**

# RA: Iterator / "for-each" semantics

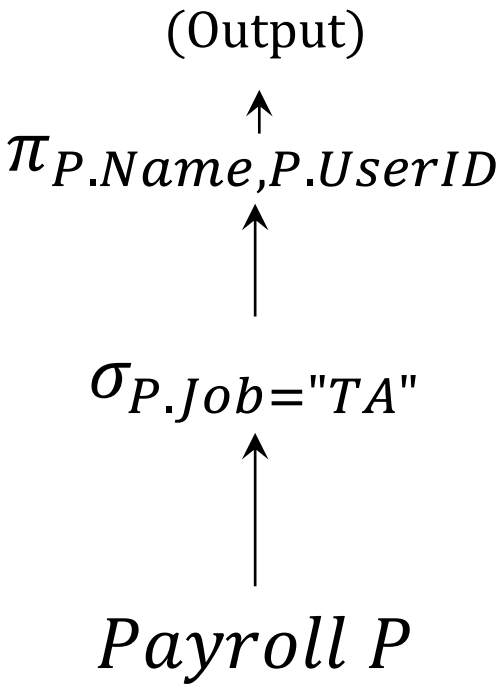
```
SELECT P.Name, P.UserID
  FROM Payroll AS P
 WHERE P.Job = 'TA';
```




Tuples "flow" up the query plan, getting filtered and modified

# RA: Iterator / "for-each" semantics

```
SELECT P.Name, P.UserID
  FROM Payroll AS P
 WHERE P.Job = 'TA';
```




**For-each semantics**

```
for each row in P:
  if (row.Job == 'TA'):
    output (row.Name, row.UserID)
```

# Iterator / "for-each" semantics

**Payroll**

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

```
for each row in P:
  if (row.Job == 'TA'):
    output (row.Name, row.UserID)
```

| Name | UserID |
|---|---|

```
SELECT P.Name, P.UserID
  FROM Payroll AS P
 WHERE P.Job = 'TA';
```

# Iterator / "for-each" semantics

**Payroll**

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

```
for each row in P:
  if (row.Job == 'TA'):
    output (row.Name, row.UserID)
```

| Name | UserID |
|---|---|
| Jack | 123 |

```
SELECT P.Name, P.UserID
  FROM Payroll AS P
 WHERE P.Job = 'TA';
```

# Iterator / "for-each" semantics

**Payroll**

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

```
for each row in P:
  if (row.Job == 'TA'):
    output (row.Name, row.UserID)
```

| Name | UserID |
|---|---|
| Jack | 123 |

```
SELECT P.Name, P.UserID
  FROM Payroll AS P
 WHERE P.Job = 'TA';
```

# Iterator / "for-each" semantics

**Payroll**

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

```
for each row in P:
  if (row.Job == 'TA'):
    output (row.Name, row.UserID)
```

| Name | UserID |
|---|---|
| Jack | 123 |
| Allison | 345 |

```
SELECT P.Name, P.UserID
  FROM Payroll AS P
 WHERE P.Job = 'TA';
```

# Iterator / "for-each" semantics

**Payroll**

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

| Name | UserID |
|---|---|
| Jack | 123 |
| Allison | 345 |

```
for each row in P:
  if (row.Job == 'TA'):
    output (row.Name, row.UserID)
```

```
SELECT P.Name, P.UserID
  FROM Payroll AS P
 WHERE P.Job = 'TA';
```

# Iterator / "for-each" semantics

**Payroll**

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

```
for each row in P:
  if (row.Job == 'TA'):
    output (row.Name, row.UserID)
```

| Name | UserID |
|---|---|
| Jack | 123 |
| Allison | 345 |

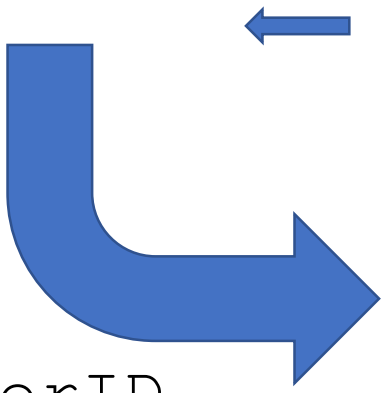
```
SELECT P.Name, P.UserID
  FROM Payroll AS P
 WHERE P.Job = 'TA';
```

# Iterator / "for-each" semantics

**Payroll**

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

```
for each row in P:
  if (row.Job == 'TA'):
    output (row.Name, row.UserID)
```



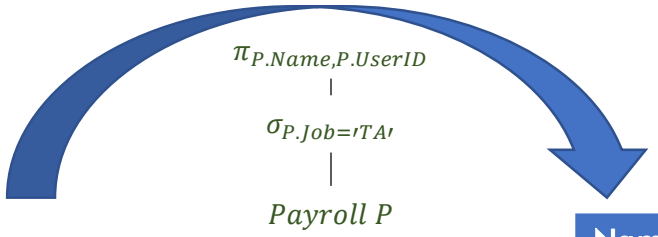
| Name | UserID |
|---|---|
| Jack | 123 |
| Allison | 345 |

```
SELECT P.Name, P.UserID
  FROM Payroll AS P
 WHERE P.Job = 'TA';
```

# Recap – SQL and RA

- SQL (Next few lectures)
  - "What data do I want"
- RA (After SQL)
  - "How do I get the data"

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |




```
SELECT P.Name, P.UserID
  FROM Payroll AS P
 WHERE P.Job = 'TA';
```

| Name | UserID |
|---|---|
| Jack | 123 |
| Allison | 345 |

# Think About This

SQL = What to compute

RA = How to compute