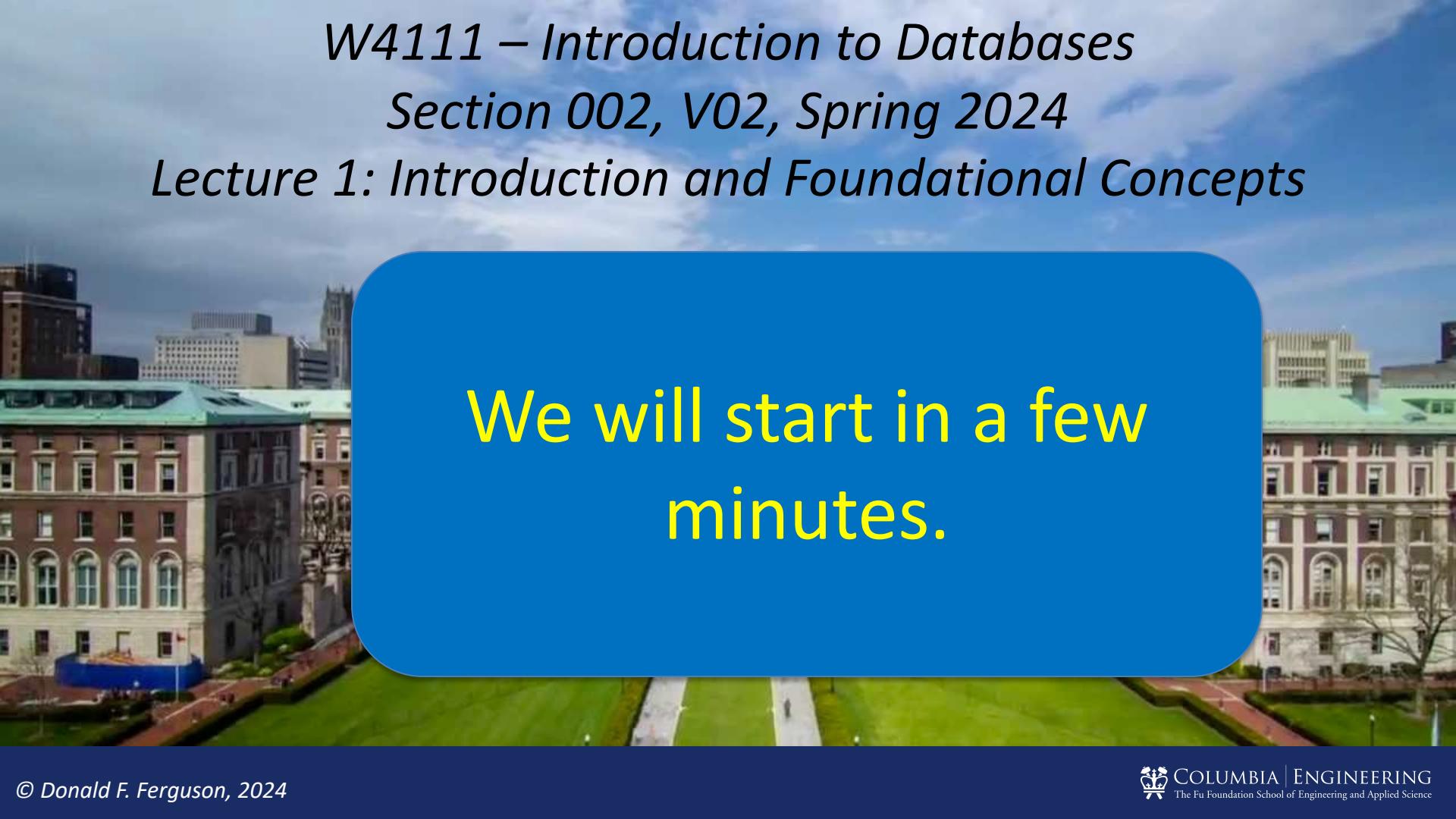


*W4111 – Introduction to Databases
Section 002, V02, Spring 2024
Lecture 1: Introduction and Foundational Concepts*



*W4111 – Introduction to Databases
Section 002, V02, Spring 2024
Lecture 1: Introduction and Foundational Concepts*



We will start in a few minutes.

W4111 Introduction to Databases:

*Faculty do not manage waitlists
for some courses, including W4111.*

*The academic admin staff in the
CS Department manages the waitlist,
priorities and enrollment.*

*You should contact advising email:
ug-advising, ms-advising, or phd-advising
@cs.columbia.edu*

Well-Being Announcement

*No peanuts, nuts, nut or peanut products,
etc. in the classroom, office hours,*

Today's Contents

Contents

- Introduction
 - Logistics, about your instructor, OHs, IAs.
 - Homework, exams.
 - ~~Data, Databases, Database Management Systems, Applications~~
 - Motivating Examples:
 - Interactive, web application.
 - Data analysis and visualization.
 - Database design, Entity-Relationship Model (Part 1)
 - Database design process.
 - ER-Model and diagrams.
 - The theory: The *Relational Model* (Part 1)
 - Relational model, schema, keys, schema diagrams.
 - Basics of relational algebra.
 - The realization: Structured Query Language (SQL) (Part 1)
 - Basics of Data Definition Language.
 - Basics of Data Manipulation Language (Query).
 - Homework 1 – (Initial) Definition and discussion.
- The material is mostly
 - History of databases
 - Motivation for databases
 - Terms
 - Concepts
 - Covering in lecture is not a good use of time.
 - Just read slides that come with book.
<https://www.db-book.com/db7/slides-dir/index.html>
 - Homework assignments and exams will test knowledge and if you have reviewed material.

Introduction

Logistics

Waitlist

- The administrative staff of the Dept. of Computer Science **directly** manages enrollment in the W4111 sections and the wait lists. This is true of several courses.
 - COMS W4111 is **ALWAYS** oversubscribed and has huge wait lists.
 - The course is a requirement for several majors and tracks.
 - There are **complex rules** about prioritizing students for course enrollment.
- Faculty follow department policies and do not manage waitlists, despite the info that identifies the waitlist as “Instructor Managed.”
- Slide 3 provides more details.

Lecture Format, Recitation, Office Hours

- Sessions:
 - Lecture: (In-Person) Lecture: Friday, 10:10AM to 12:40PM (TBD).
 - Recitation: Periodic, optional, online recitations
 - Either in-person/online or just online. Will announce in advance.
 - Usually held when there is student interest, homework due, exams, etc.
- Office hours:
 - In-person: Friday: 8:30AM – 10:00 AM (488 Computer Science Building); Sometimes also 1pm-3pm, Friday, 488 CSB.
 - Extra office hours: I hold a lot of extra office hours, usually based on workload around assignment due dates and exams.
- Collaboration/contact:
 - We will use Ed Stem, which is available from CourseWorks.
You can get access from the side menu on the CourseWorks page for the class.
 - I am usually monitoring [Slack](#). Subscribe to the channel #w4111S24.
 - The course lectures, sample code, etc. will be in a [GitHub](#) repository.
 - The course [website](#) provides additional information.
 - There will be a [course calendar](#) with OHs, assignments, exams,
- **Note: Your health, safety and well-being are **ALWAYS** my primary concern.**
Speak to me if you need special considerations and I will do the best that I can.

About Your Instructor

About your Instructor

- 40 years in computer science industry:
 - IBM Fellow
 - Microsoft Technical Fellow
 - Chief Technology Officer, CA technologies
 - Dell Senior Technical Fellow.
 - CTO, Co-Founder, [Seeka.tv](https://seeka.tv)
 - Ansys Fellow
- Academic experience:
 - BA, MS, Ph.D., Computer Science, Columbia University
 - Approx. 18 semesters as an Adjunct Professor
 - Fulltime Professor of Professional Practice in CS (2018)
 - Courses:
 - E1006: Intro. to Computing
 - W4111: Intro. to Databases
 - E6998, E6156: Advanced Topics in SW Engineering (Cloud Computing)
- Approx. 65 technical publications; Approx. 12 patents



Personal:

- Two children:
 - College student
 - 2019 Barnard Graduate
- Hobbies:
 - Krav Maga, Black Belt in Kenpo Karate
 - Former 1LT, [New York Guard](https://newyorkguard.org)
 - Bicycling
 - Astronomy
- Languages:
 - Proficient in Spanish
 - Learning Arabic

About the Course

The Course

- From the new/pending Columbia University Bulletin

"Prerequisites: COMS W3134, COMS W3136, or COMS W3137; or the instructor's permission.

The course covers what a database system is, how to design databases effectively and in a principled manner, how to query databases, and how to develop applications using databases: entity-relationship modeling, logical design of relational databases, relational algebra, SQL, database application development, database security, and an overview of query optimization and transaction processing. Additional topics generally include NoSQL, graph, object-relational, and cloud databases, as well as data preparation and cleaning of real-world data. The course offers both programming and non-programming paths for homework and projects, to accommodate students with different programming skills and backgrounds."

- Prerequisites:
 - COMS W3134, COMS W3136, or COMS W3137 are data structures classes. All of these courses require extensive programming, in Java.
 - A course in data structures is helpful for this section of W4111 but not essential. I waive the requirement.
 - We will help you with any data structures knowledge you lack.
- Programming in/for this class:
 - There will be a "non-programming" option, which we will discuss below.
 - For students who want to take the programming track, we will use Python. I will provide motivation for choosing Python below.

Course Objectives

- Have fun, learn a lot and come to appreciate and enjoy some amazing technology. Data and databases have and will change the world.
- Provide a foundation that allows you to succeed in future courses. This is an *introduction* to databases. The technology is crucial for future courses
 - Big data, data analysis, data science
 - Advanced database classes
 - Machine learning
 - Numerical and data analytics in operations research, engineering, economics, finance, life sciences, financial engineering, medicine, etc.
- Enable you to successfully apply the technology in your work and profession.

Have cool stuff to talk about on interviews and in your resumes.

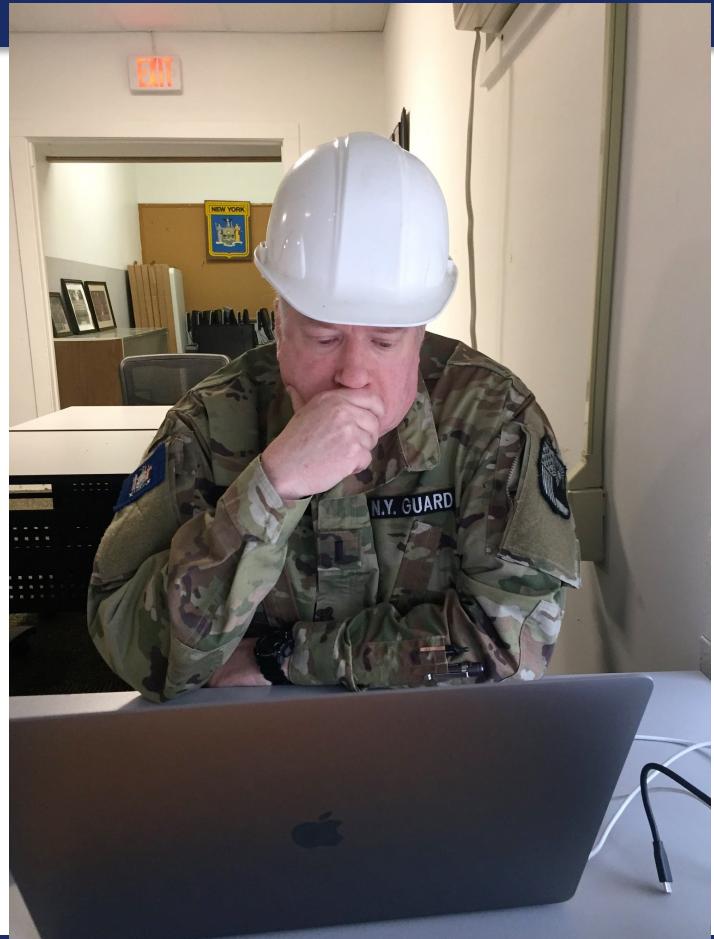
The Course – Value and my Perspective

- This course is foundational, and will teach you the core concepts in
 - Data modeling
 - Data model implementation; Data manipulation.
 - Different database models and database management systems.
 - Implementation and architecture of data centric applications and database management systems.
- ANY non-trivial application
 - Requires a well-designed data model.
 - Implements a data model and manipulates data.
 - Uses a database management system.
- Understanding databases and database management is core to the “hottest fields” in computer science, e.g.
 - Data science
 - Machine learning
 - Intelligent (Autonomous) systems
 - Internet-of-Things
 - Cybersecurity
 - Cloud Computing
- Database, database application, etc. skills are increasingly central to many disciplines, including:
 - Economics.
 - IEOR, financial engineering.
 - Mechanical and electrical engineering.
 - Medicine, pharmaceuticals
 - ...
- University courses on databases sometimes focus on theory and abstract concepts. This course will cover theory, but there will be an increased emphasis on:
 - Practical, hands-on applications of databases.
 - Patterns and best practices.
 - Developing and understanding database centric applications.
 - Using databases and various tools for data analysis, visualization and insight.
- **Personal perspective**
 - A large percent of my career has been spent figuring out or leading teams that figured out how to model, implement and manipulate data.
 - I have used the information in this class more than anything else I have learned.
 - This will likely be true for you.

Surprising Example

Example: This is a photo of me using a database during COVID-19 mobilization.

- Match service members
 - To JTF-HQ requests for personnel
 - Based on assignment needs
 - And service member
 - Skills
 - Availability
 - Tracked in a DBMS.
 - The hardhat is because DB usage can be very dangerous 😱.
-
- No joking: I had to build an application that used:
 - Relational DBMS.
 - Python, Jupyter Notebook.
 - Google Sheets with Apps Script.
 - Google Forms.
 - This course's technology and these skills are surprisingly applicable.



Modules

Each section of W4111 is slightly different based on student interest and professor's focus. There is a common, core syllabus. Professors cover topics in different orders and grouping based on teaching style. This section of W4111 has four modules:

- **Foundational concepts (50% of semester):** This module covers concepts like data models, relational model, relational databases and applications, schema, normalization, ... The module focuses on the relational model and relational databases. The concepts are critical and foundational for all types of databases and data centric applications.
- **Database management system architecture and implementation (10%):** This module covers the software architecture, algorithms and implementation techniques that allow [databases management systems](#) to deliver functions. Topics include memory hierarchy, storage systems, caching/buffer pools, indexes, query processing, query optimization, transaction processing, isolation and concurrency control.
- **NoSQL – “Not Only SQL” databases (20%):** This module provides motivation for [“NoSQL”](#) data models and databases, and covers examples and use cases. The module also includes cloud databases and databases-as-a-service.
- **Data Enabled Decision Support (20%):** This module covers data warehouses, data import and cleanse, OLAP, Pivot Tables, Star Schema, reporting and visualization, and provides an overview of analysis techniques, e.g. clustering, classification, analysis, mining.

Environment

Course Resources and Development Environment

- Recommended textbook:
 - *Database System Concepts. Seventh Edition.* (ISBN 9780078022159)
 - There is a website associated with the textbook: <https://www.db-book.com/>. The site has:
 - Slides for each chapter.
 - Example data.
 - Textbooks are expensive. You can easily get through the course using website, lecture material,
- Install a new, most recent, isolated/single user instance of Anaconda environment.
 - **Install just for yourself and within your home directory.**
 - You must install the most recent version for Python 3.
 - You can isolate the new instance from other instances to avoid conflicts, or set up custom environments.
- Development environments: Students are entitled to a free, annual [JetBrains professional license](#).
 - Students have had problems with MySQL Workbench. We will use [DataGrip](#). Please install.
 - Installing [PyCharm](#) is recommended for all and required for the programming track.
- Install [MySQL Server Community Edition](#).
 - When prompted, choose [legacy authentication](#) method.
 - Set your root password to **dbuserdbuser**. (**Remember your user name and password**)

*Homework
Exams
Grading*

Assignments, Exams, Grading

- Point value of assignments and exams
 - 50%: Homework assignments:
 - 5 HWs, approximately one HW every two-three weeks.
 - Each is worth 10. We will sometimes break a HW into a couple of subparts.
 - Homework format:
 - Common to all tracks:
 - Questions requiring written answers and diagrams.
 - Implement/execute/test various database operations required to solve a use case/question.
 - Format will be an iPython/Jupyter Notebook.
 - Track specific: Incremental development of a project:
 - Programming track – web application.
 - Non-programming track – data analysis/visualization.
 - You receive a total of **five** late submission days that you can use during the semester.
- Exams: Both the midterm and final have an in-class and take-home element.
 - 20% of grade is midterm exam score.
 - 30% is final exam score.
- 94 -100 points is **an A**. There are opportunities to get extra-credit.

To Program or Not To Program, ...

- From the department's guidance for the course:
 - "To accommodate the diverse backgrounds of the students who take this class, all sections of the class should include a non-programming option for projects and assignments. We (NOTE: Does not include me) have successfully offered such an option in some sections of the class for many years: students can either program a web application to interface with a DB of their own design, or alternatively follow the non-programming option and come up with a quantifiably more detailed DB design/data."
- What does not constitute programming?
 - Writing queries for a relational, graph, document, or other data management system does not constitute programming and can be expected of all students.
- What constitutes programming?
 - Reading more than a handful (1-5) lines of code, Writing Python code that is not directly required to write a query."
- Why I previous only did a programming track:
 - Any non-trivial use of data and databases in any field requires programming. This is not Star Trek.
You cannot shout, "Computer! Analyze ..."
 - You might think, "But, I do not want to program. I want to go into financial services, private equity, medicine, ... I can just use tools like Pandas or Tableau". You will do some programming in these jobs for complex scenarios. Get over it.
- Tracks: Programming and non-programming.
 - There is a common core that involves understanding concepts, modeling data, using databases, etc.
 - Exams are the same for both tracks. Mix of written questions and practical exercises.
 - Homework assignments: There is a common core on all assignments for both tracks. Additionally,
 - Programming track will incrementally build a database centric, cloud-based web application.
 - Non-programming track will do a more complex, database centric project.

Homework Assignments

- My homework assignments are:
 - Open ended.
 - Vaguely specified.
- You will complain. I will listen sympathetically.
 - You will savage me on the instructor reviews and CULPA.
 - My professional colleagues and I will laugh at you behind your back.
- Management, clients, partners, etc.
 - Do not understand technology as well as we do.
 - You will get requirements like, "I want it colored mauve because that has more RAM."
- Converting vague requests into a useful, meaningful project that we can implement is what we do. Get over it.

You are senior people at a top school. You need to define concrete solutions and approaches from ambiguity. No one showed Thomas Edison a light bulb and then said, "Build this."

Donald Ferguson on 2019-01-29

Omonbude Emmanuel
@BUDESCODE

To replace programmers with
Robots, clients will have to
accurately describe what they
want.
We're safe.

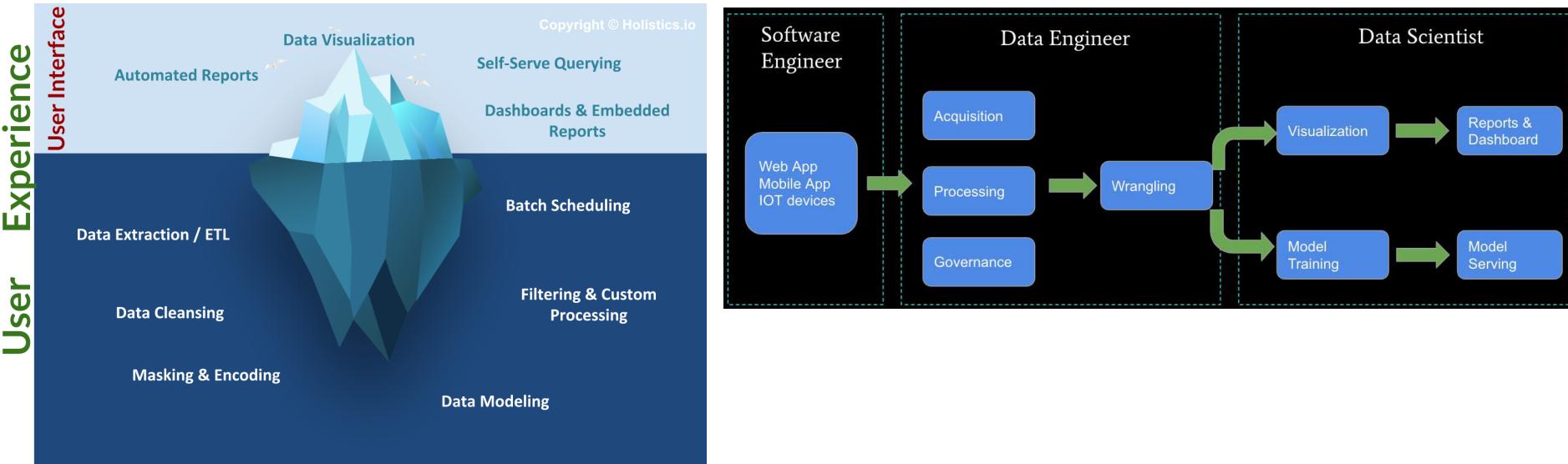
7:25 PM · 20 Jul 20 · Twitter for Android

Motivating Examples

Two Common Database Applications

- Operational/Interactive:
 - Users and roles can create, retrieve, update, search and delete “records.”
 - Examples: SSOL, ATMs,
- Business Intelligence, Decision Support, ...:
 - Users can perform complex queries and analyze a lot of data to generate a report, make a decision,
 - Examples: Build AI/ML training data, dashboards,
- One of our major examples this semester will be IMDB and Game of Thrones:
 - IMDB: <https://developer.imdb.com/non-commercial-datasets/>
 - Game of Thrones: <https://developer.imdb.com/non-commercial-datasets/>
- We will build a simple web application and do some data engineering.

Business Intelligence, Insight, Analysis,



- The “fun” stuff in data science and AI/ML is the “tip of the iceberg.”
- Data engineering is a necessary condition for producing analyzable data. This is often more than 80% of the hard work.
- We will do some small data engineering projects in this course.

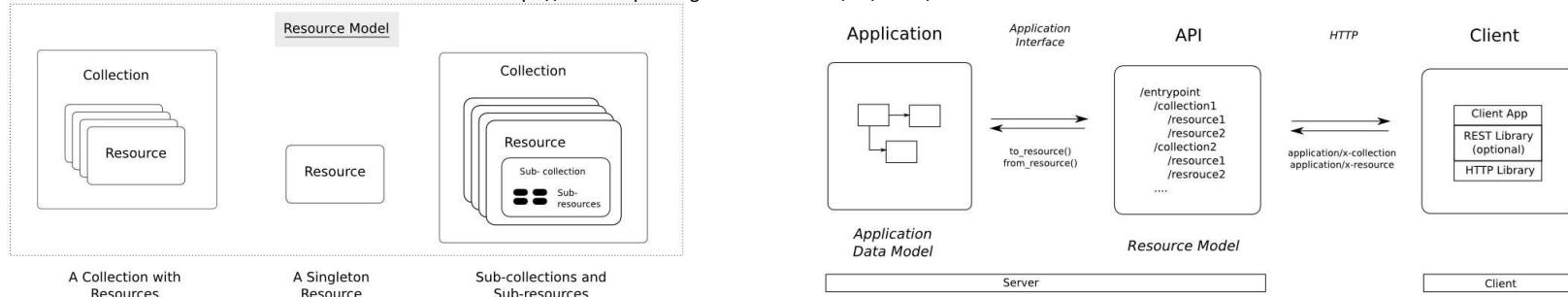
Switch to Notebook

- /Users/donaldferguson/Dropbox/000/000-Columbia/2024-Spring/W4111-Intro-to-Databases-Spring-2024/Lectures/s-2024-Lecture-1/s-2024-w4111-examples.ipynb
- Skip for now.

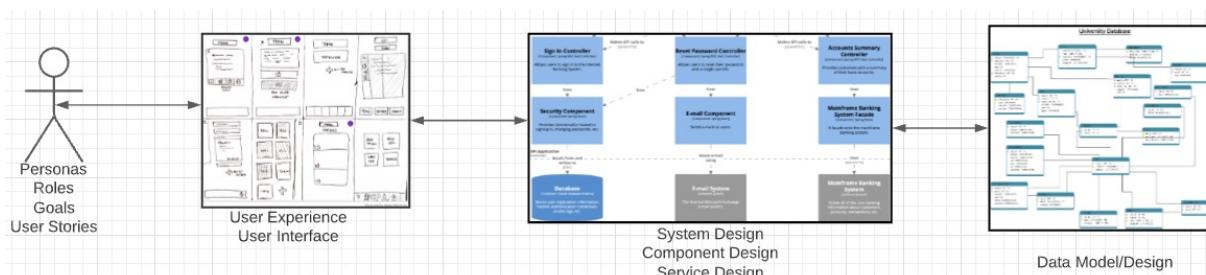
Web Application Problem Statement

- We must build a system that supports create, retrieve, update and delete for IMDB and Game of Thrones Datasets.
- This requires implementing *create, retrieve, update and delete (CRUD)* for resources.

<https://restful-api-design.readthedocs.io/en/latest/resources.html>



- We will design, develop, test and deploy the system iteratively and continuously.
- There are four core domains.

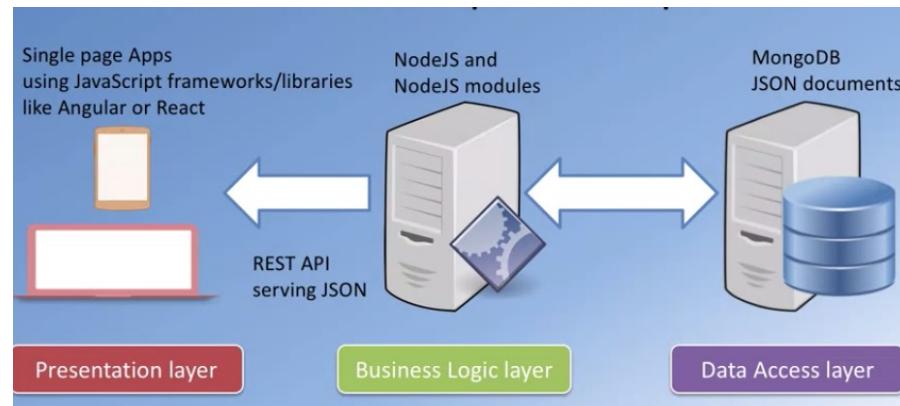


- In this course,
- We focus on the data dimension.
- We will get some insight into the other dimensions.

Interactive/Operational

- “A full stack web developer is a person who can develop both client and server software. In addition to mastering HTML and CSS, he/she also knows how to:
 - Program a browser (like using JavaScript, jQuery, Angular, or Vue)
 - Program a server (like using PHP, ASP, Python, or Node)
 - Program a database (like using SQL, SQLite, or MongoDB)”
- We will do a simple full stack app.

- Three databases:
 - MySQL
 - MongoDB
 - Neo4j
- The application tier will be Python and FastAPI.
- The web UI will be Angular.
- The primary focus is the data layer and application layer that access it.
- I will provide a simple UI and template.



Interactive Full Stack Application

- Show “code:”
 - PyCharm:
`/Users/donaldferguson/Dropbox/000-NewProjects/W4111-FastAPI-IMDB-Solution`
 - WebStorm:
`/Users/donaldferguson/Dropbox/000-NewProjects/current-dashboard`
 - DataGrip:
`s23_got_clean,`
`S23_Classic_Models`Skip for Now.
- Show execution:
 - Browser, Postman, Notebook:
<http://0.0.0.0:8001/>, <http://0.0.0.0:8001/docs>,
<http://0.0.0.0:8001/api/characters/sansa%20stark>
 - Angular Application:
<http://localhost:4200/#/character>
<http://localhost:4200/#/customer>

Approach for First Few Lectures

Lectures and Topics

► Chapter 1 Introduction

▼ PART ONE RELATIONAL LANGUAGES

► Chapter 2 Introduction to the Relational M...

► Chapter 3 Introduction to SQL

► Chapter 4 Intermediate SQL

► Chapter 5 Advanced SQL

▼ PART TWO DATABASE DESIGN

► Chapter 6 Database Design Using the E-R ...

► Chapter 7 Relational Database Design

▼ PART THREE APPLICATION DESIGN AND DEV...

► Chapter 8 Complex Data Types

- Chapter 1:

- Is interesting, but something easily learned from reading slides, docs, etc.
 - **You are responsible for reading the slides and independent study.**

- Books and standard syllabuses tend to be sequential.

Cover and complete topics one at a time.

- My view is that there is a core conceptual model with three realizations:

- Entity, relationships,; the concepts.
 - ER design and modeling.
 - Implementation:
 - Relational model/algebra.
 - SQL.
 - Resource/REST oriented.
 -

- My approach is to incrementally and iteratively:

- Introduce concepts.
 - Explain the realizations.
 - Because implementing a real system requires the approach of concept, design, implement in several layers.

Database Modeling, ER Modeling (I)



Design Phases

- Initial phase -- characterize fully the data needs of the prospective database users.
- Second phase -- choosing a data model
 - Applying the concepts of the chosen data model
 - Translating these requirements into a conceptual schema of the database.
 - A fully developed conceptual schema indicates the functional requirements of the enterprise.
 - Describe the kinds of operations (or transactions) that will be performed on the data.

DFF Comments:

- We see slides with this formatting, they come directly from the presentations associated with the textbook. (<https://www.db-book.com/db7/slides-dir/index.html>)
- The number at the bottom is of the form chapter.slide_no.
- I try to put my comments, modifications and annotations in red text, or inside a red rectangle/callout.



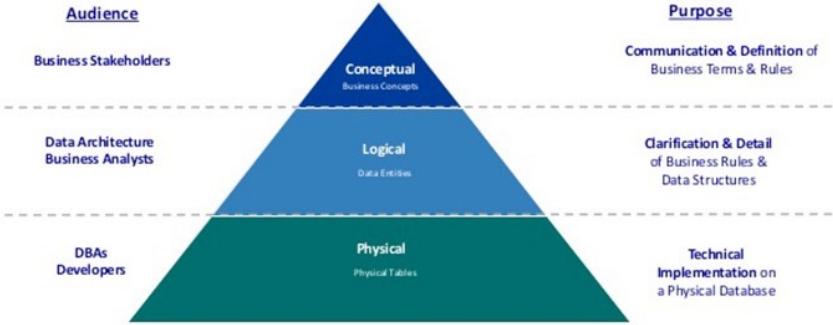
Design Phases (Cont.)

- Final Phase -- Moving from an abstract data model to the implementation of the database
 - Logical Design – Deciding on the database schema. Database design requires that we find a “good” collection of relation schemas.
 - Business decision – What attributes should we record in the database?
 - Computer Science decision – What relation schemas should we have and how should the attributes be distributed among the various relation schemas?
 - Physical Design – Deciding on the physical layout of the database

A Common and my Approach: Conceptual → Logical → Physical

<https://ehkioya.com/conceptual-logical-physical-database-modeling/>

Levels of Data Modeling

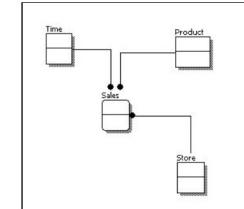


- It is easy to get carried away with modeling. You can spend all your time modeling and not actually build the schema.
- We will use the approaches in class.
- Mostly to understand concepts and patterns.

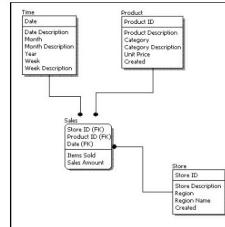
<https://www.1keydata.com/datawarehousing/data-modeling-levels.html>

Feature	Conceptual	Logical	Physical
Entity Names	✓	✓	
Entity Relationships	✓	✓	
Attributes		✓	
Primary Keys		✓	✓
Foreign Keys		✓	✓
Table Names			✓
Column Names			✓
Column Data Types			✓

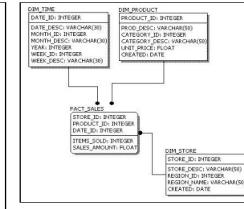
Conceptual Model Design



Logical Model Design



Physical Model Design



<https://www.1keydata.com/datawarehousing/data-modeling-levels.html>



ER model -- Database Modeling

- The ER data mode was developed to facilitate database design by allowing specification of an **enterprise schema** that represents the overall logical structure of a database.
- The ER data model employs three basic concepts:
 - entity sets,
 - relationship sets,
 - attributes.
- The ER model also has an associated diagrammatic representation, the **ER diagram**, which can express the overall logical structure of a database graphically.



Entity Sets

COMS W4111 002 01 2024

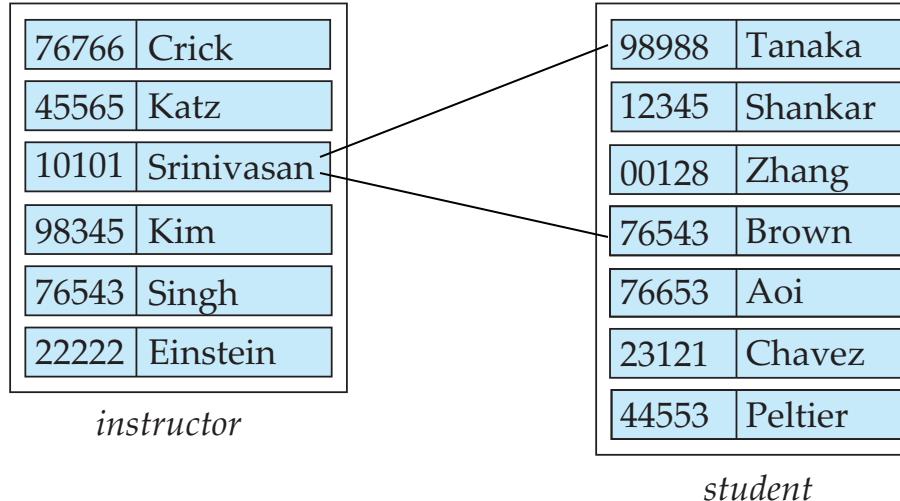
- An **entity** is an object that exists and is distinguishable from other objects.
 - Example: specific person, company, event, plant
- An **entity set** is a set of entities of the **same type** that share the same properties.
 - Example: set of all persons, companies, trees, holidays
- An entity is represented by a set of attributes; i.e., descriptive properties **possessed by all** members of an entity set.
 - Example:
 $\text{instructor} = (\text{ID}, \text{name}, \text{salary})$
 $\text{course} = (\text{course_id}, \text{title}, \text{credits})$
- A subset of the attributes form a **primary key** of the entity set; i.e., uniquely identifying each member of the set.

DFF Comments:

- Some of these statements apply primarily to OO systems and the relational/SQL models.
- A motivation for “No SQL” is to relax the constraints.



Entity Sets -- *instructor* and *student*



(10101, 98988)
(10101, 76543)



Relationship Sets

- A **relationship** is an association among several entities

Example:

44553 (Peltier) advisor 22222 (Einstein)
student entity relationship set instructor entity

- A **relationship set** is a mathematical relation among $n \geq 2$ entities, each taken from entity sets

$$\{(e_1, e_2, \dots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \dots, e_n \in E_n\}$$

where (e_1, e_2, \dots, e_n) is a relationship

- Example:

$$(44553, 22222) \in \text{advisor}$$

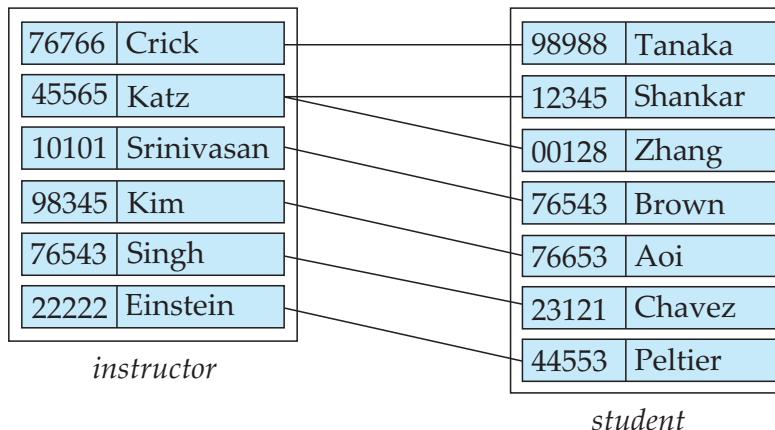
DFF Comments:

- Nobody thinks about relationships this way.
- There is no idea so simple that a DB professor cannot make it confusing, usually by using math.



Relationship Sets (Cont.)

- Example: we define the relationship set *advisor* to denote the associations between students and the instructors who act as their advisors.
- Pictorially, we draw a line between related entities.



DFF Comment:

- In this diagram, the lines are the relationship set.
- Many DB models use a 3rd entity set to represent complex relationships.

DFF Comments:

- Nobody draws the diagrams this way, but ...
- Sometimes thinking this way helps understand other ways to depict the concept.



Representing Entity sets in ER Diagram

- Entity sets can be represented graphically as follows:
 - Rectangles represent entity sets.
 - Attributes listed inside entity rectangle
 - Underline indicates primary key attributes

<i>instructor</i>
<u>ID</u>
<i>name</i>
<i>salary</i>

<i>student</i>
<u>ID</u>
<i>name</i>
<i>tot_cred</i>

“Visual modeling is the use of semantically rich, graphical and textual design notations to capture software designs. A notation, such as UML, allows the level of abstraction to be raised, while maintaining rigorous syntax and semantics. In this way, it improves communication in the design team, as the design is formed and reviewed, allowing the reader to reason about the design, and it provides an unambiguous basis for implementation.”

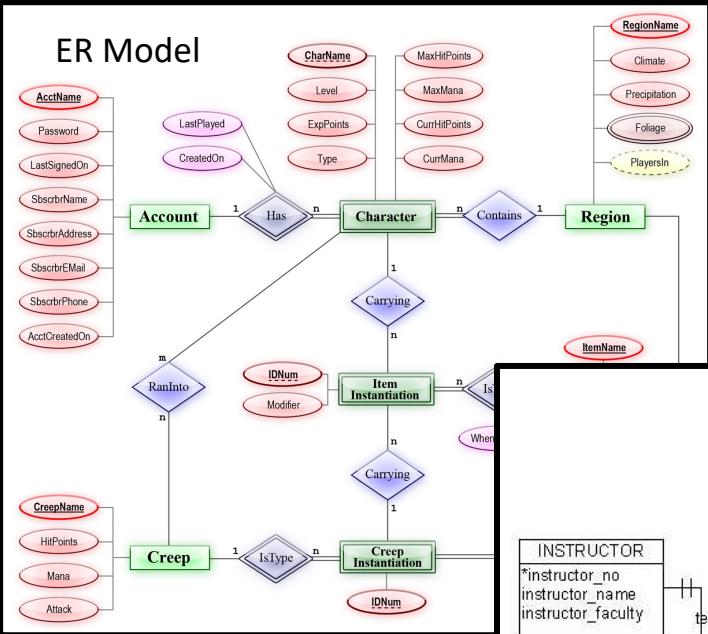
Instructor	
PK	ID
	<i>salary</i>
	<i>name</i>

Crow's
Foot
Notation

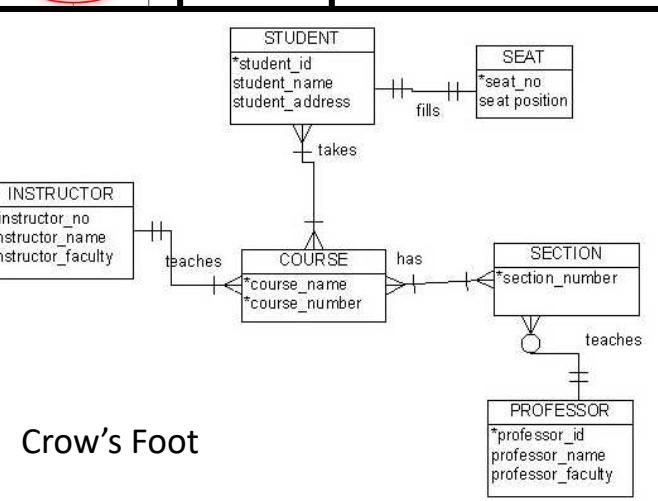
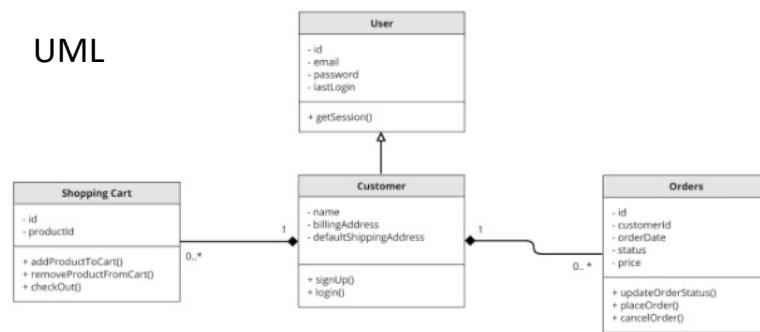
Student	
PK	ID
	<i>name</i>
	<i>tot_cred</i>

Visual Notation – Many Notations

ER Model



UML

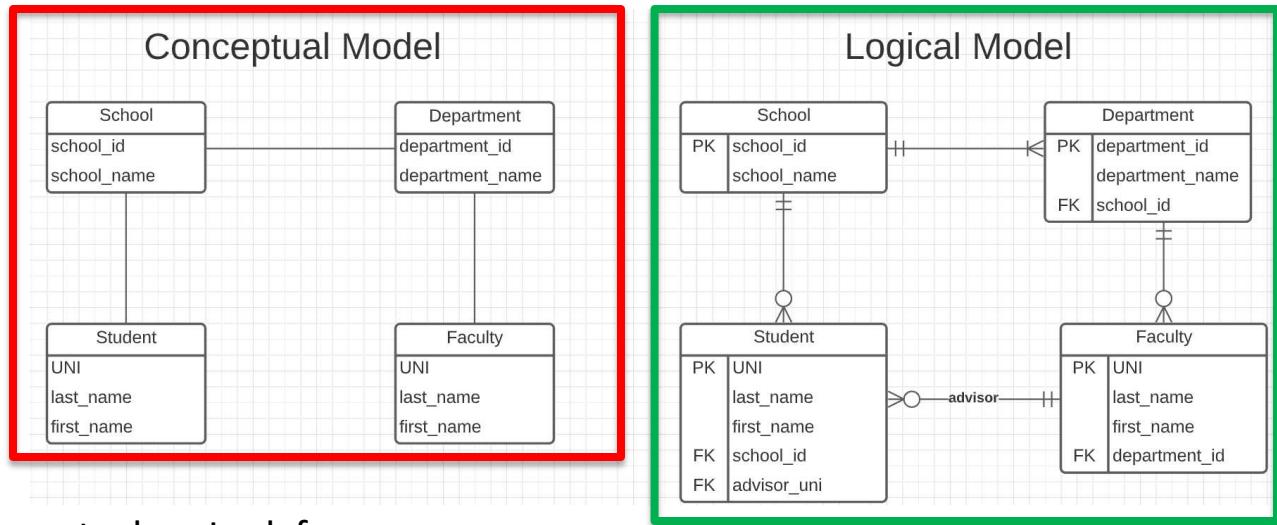


Crow's Foot

- “Other,” i.e. PowerPoint is the most common modeling notation.
- It is easy to get “carried away.”
- The trick is to do “just enough modeling.”
- I mostly use Crow’s Foot
 - It is “just enough”
 - But lacks some capabilities.
- The book uses ER notation.

Do a Slightly More complex University Database ER Model

- The model has:
 - Four entity sets:
 - School*
 - Department*
 - Student*
 - Faculty*
 - Three relationship sets:
 - Student-School*
 - School-Department*
 - Faculty-Department*.

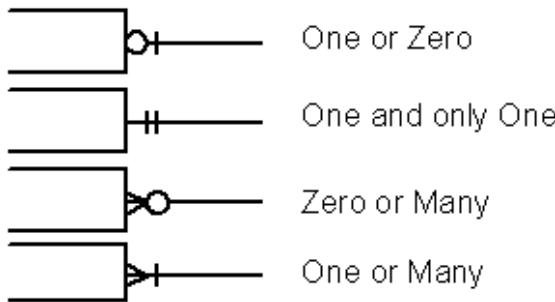


- This is the level of detail I want when I ask for:
 - Conceptual Model diagram.
 - Logical Model diagram.
- Some online tools with “free,” constrained usage.
 - Lucidchart (<https://www.lucidchart.com/>)
 - Vertabelo (<https://vertabelo.com/>)

Notation has Precise Meaning

- Attribute annotations:
 - PK = Primary Key
 - FK = Foreign Key
- Line annotations:
 - We will spend a lot of time discussing keys.
 - We will start in a couple of slides.

Summary of Crow's Foot Notation



- We will learn over time and there are good tutorials (<https://www.lucidchart.com/pages/er-diagrams>) to help study and refresh.

What Does this Mean? Let's Get Started

Primary Key means that the value occurs at most once.

This is a statement about the domain, not the actual data currently in the table.

School Code	School Name
CC	Columbia College
SEAS	Fu Foundation School of Engineering and Applied Science
GSAS	Graduate School of Arts and Sciences
GS	General Studies
....

Foreign Key means that if a value occurs in `school_id` for any row, there must be a row in School with that key.

UNI	Last name	First name	school_id
dff9	Ferguson	Donald	CC
js11	Smith	John	GS
jp9	Public	James	CC
bb101	Baggins	Bilbo	CC
....

The line notations mean:

- A student is related to EXACTLY ONE school.
- A School may be related to 0, 1 or many students.



ER model -- Database Modeling

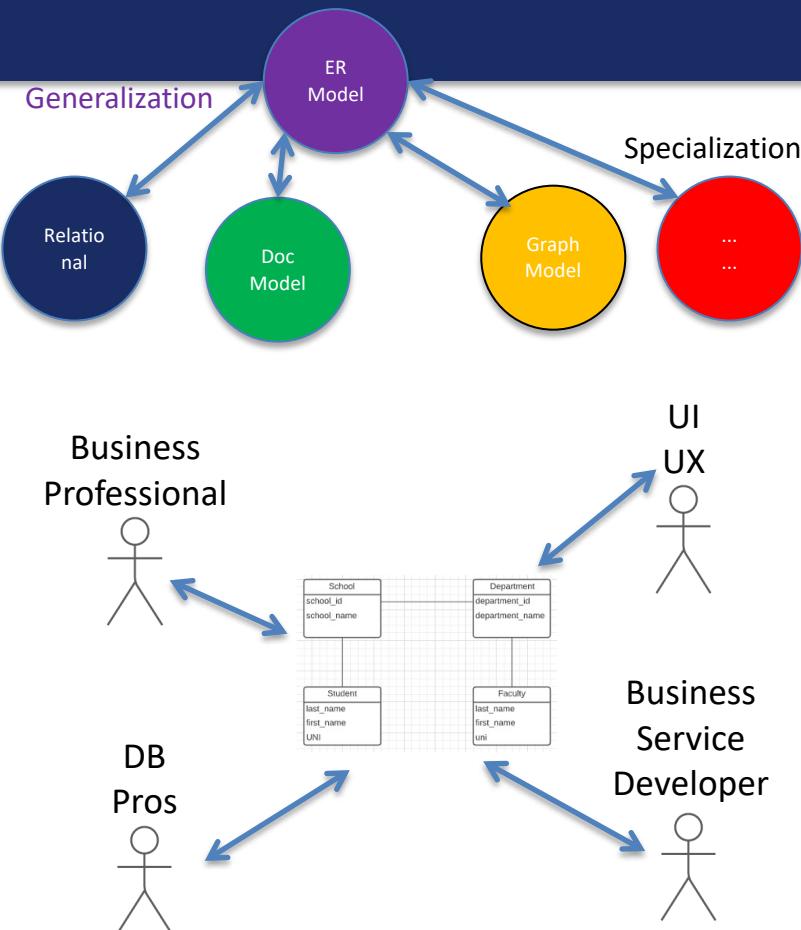
- The ER data mode was developed to facilitate database design by allowing specification of an **enterprise schema** that represents the overall logical structure of a database.
- The ER data model employs three basic concepts:
 - entity sets,
 - relationship sets,
 - attributes.
- The ER model also has an associated diagrammatic representation, the **ER diagram**, which can express the overall logical structure of a database graphically.

DFF Comments:

- The book and slides do not do a great job of motivating the ER model or ER diagrams.
- Why do people and teams think about or use the ER model and modeling?

ER Model and ER Modeling

- ER Model: Agility, Separation of Concerns
 - ER model is a generalization that most DB models implement in some form.
 - Using the ER model enables:
 - Thinking about and collaborating on design with getting bogged down in details.
 - Enable flexible choices about how to realize/Implement data.
- ER Diagrams: Communication, Quality, Precision
 - With a little experience, everyone can understand and ER diagram.
 - Easier to discuss and collaborate on application's data than showing SQL table definitions, JSON,
 - People think visually. That is why we have whiteboards. ER diagrams are precise and unambiguous.
 - Guides you to think about relationships, keys, ... And prevents “re-dos” later in the process. It is easier to fix a diagram than a database schema.



ER Modeling – Reasonably Good Summary

Advantages of ER Model

Conceptually it is very simple: ER model is very simple because if we know relationship between entities and attributes, then we can easily draw an ER diagram.

Better visual representation: ER model is a diagrammatic representation of any logical structure of database. By seeing ER diagram, we can easily understand relationship among entities and relationship.

Effective communication tool: It is an effective communication tool for database designer.

Highly integrated with relational model: ER model can be easily converted into relational model by simply converting ER model into tables.

Easy conversion to any data model: ER model can be easily converted into another data model like hierarchical data model, network data model and so on.

Disadvantages of ER Model

Limited constraints and specification

Loss of information content: Some information be lost or hidden in ER model

Limited relationship representation: ER model represents limited relationship as compared to another data models like relational model etc.

No representation of data manipulation: It is difficult to show data manipulation in ER model.

Popular for high level design: ER model is very popular for designing high level design

No industry standard for notation

<https://pctechnicalpro.blogspot.com/2017/04/advantages-disadvantages-er-model-dbms.html>

Note:

- If you get to use Google to help with take home exams, HW, etc.
- I get to use Google to help with slides.

Theory: Relational Model (I)



Relational Model

- All the data is stored in various tables.
- Example of tabular data in the relational model



Ted Codd
Turing Award 1981

The diagram shows a table with four columns labeled *ID*, *name*, *dept_name*, and *salary*. Two arrows point from the text "Columns" to the first two columns of the table. Another arrow points from the text "Rows" to the second column of the table, indicating that the second column contains multiple rows of data.

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

(a) The *instructor* table

- The “relation” is the “table.”
 - In my big space of pieces of data. *ID*, *name*, *dept_name*, *salary* are somehow related.
 - This causes confusion, because the ER and other models use “relation” to mean something else.
- Core concepts:
 - Relation
 - Tuple (Row)
 - Column (Attribute)



Example of a *Instructor* Relation

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Diagram annotations:

- Three arrows point from the text "attributes (or columns)" to the header row of the table.
- Two arrows point from the text "tuples (or rows)" to the second and third data rows of the table.



Attribute

- The set of allowed values for each attribute is called the **domain** of the attribute
- Attribute values are (normally) required to be **atomic**; that is, indivisible
- The special value **null** is a member of every domain. Indicated that the value is “unknown”
- The null value causes complications in the definition of many operations

Ferguson, Donald

Ferguson Donald

COMSW4111

DFF Comments:

- I will explain the importance of atomic attributes and null in examples.
- Atomic and use of Null is important!



Relations are Unordered

- Order of tuples is irrelevant (tuples may be stored in an arbitrary order)
- Example: *instructor* relation with unordered tuples

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000



Database Schema

- Database schema -- is the logical structure of the database.
- Database instance -- is a snapshot of the data in the database at a given instant in time.
- Example:
 - schema: *instructor (ID, name, dept_name, salary)*
 - Instance:

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000



Keys

- Let $K \subseteq R$
- K is a **superkey** of R if values for K are sufficient to identify a unique tuple of each possible relation $r(R)$
 - Example: $\{ID\}$ and $\{ID, name\}$ are both superkeys of *instructor*.
- Superkey K is a **candidate key** if K is minimal
Example: $\{ID\}$ is a candidate key for *Instructor*
- One of the candidate keys is selected to be the **primary key**.
 - which one?
- **Foreign key** constraint: Value in one relation must appear in another
 - **Referencing** relation
 - **Referenced** relation
 - Example: *dept_name* in *instructor* is a foreign key from *instructor* referencing *department*

Notation

Classroom relation

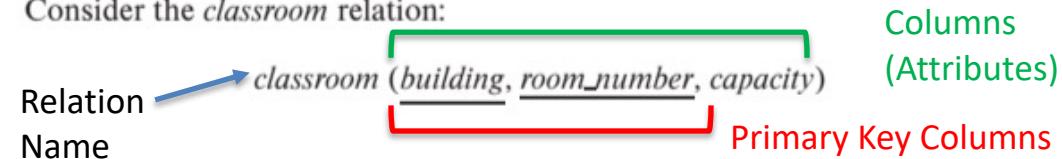
building	room_number	capacity
Packard	101	500
Painter	100	125
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

- The primary key is a *composite key*. Neither column is a key (unique) by itself.
- Keys are statements about all possible, valid tuples and not just the ones in the relation.
 - Capacity is unique in this specific data, but clearly not unique for all possible data.
 - In this domain, there cannot be two classrooms with the same building and room number.
- Relation schema:
 - Underline indicates a primary key column. There is no standard way to indicate other types of key.
 - We will use **bold** to indicate foreign keys.
 - You will sometimes see things like *classroom(building:string, room_number:number, capacity:number)*

classroom schema

It is customary to list the primary key attributes of a relation schema before the other attributes; for example, the *dept_name* attribute of *department* is listed first, since it is the primary key. Primary key attributes are also underlined.

Consider the *classroom* relation:

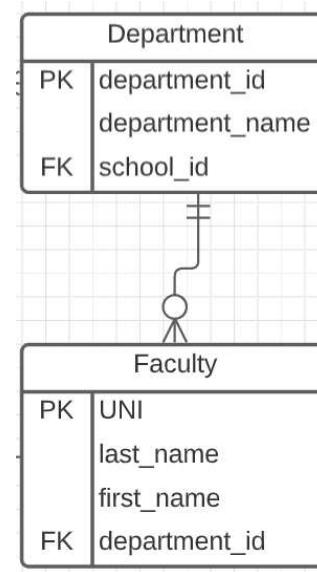


Observations

- Keys:
 - Will be baffling. It takes time and experience to understand/appreciate.
 - There are many, many types of keys with formal definitions.
 - I explain the formality but focus on the concepts and applications.
- No one uses the formal, relational model. So, why do we study it?
 - Is very helpful when understanding concepts that we cover later in the course, especially query optimization and processing.
 - There are many realizations of the model and algebra, and understanding the foundation helps with understanding language/engine capabilities.
 - The model has helped with innovating new approaches, and you may innovate in data and query models in your future.

Start Building our Datamodel

- First two relations in relational schema notation:
 - Department(department_name, department_id)
 - Faculty(UNI, first_name, last_name, **department_id**)
- Relationship:
 - A faculty “has” exactly one department.
 - A department has 0, 1 or many faculty.





Relational Query Languages

- Procedural versus non-procedural, or declarative
- “Pure” languages:
 - Relational algebra
 - Tuple relational calculus
 - Domain relational calculus
- The above 3 pure languages are equivalent in computing power
- We will concentrate in this chapter on relational algebra
 - Not turning-machine equivalent
 - Consists of 6 basic operations

DFF Comments:

- You will sometimes see other operator, e.g. \leftarrow Assignment.
- Relational algebra focuses on *retrieve*. You can sort of do Create, Update, Delete.
- The SQL Language, which we will see, extends relational algebra.



Relational Algebra

- A procedural language consisting of a set of operations that take one or two relations as input and produce a new relation as their result.
- Six basic operators
 - select: σ
 - project: Π
 - union: \cup
 - set difference: $-$
 - Cartesian product: \times
 - rename: ρ



Select Operation

- The **select** operation selects tuples that satisfy a given predicate.
- Notation: $\sigma_p(r)$
- p is called the **selection predicate**
- Example: select those tuples of the *instructor* relation where the instructor is in the “Physics” department.
 - Query

$$\sigma_{dept_name = "Physics"}(instructor)$$

- Result

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
22222	Einstein	Physics	95000
33456	Gold	Physics	87000



Select Operation (Cont.)

- We allow comparisons using
 $=, \neq, >, \geq, <, \leq$
in the selection predicate.
- We can combine several predicates into a larger predicate by using the connectives:
 \wedge (**and**), \vee (**or**), \neg (**not**)
- Example: Find the instructors in Physics with a salary greater \$90,000, we write:

$$\sigma_{dept_name = "Physics"} \wedge salary > 90,000 (instructor)$$

- Then select predicate may include comparisons between two attributes.
 - Example, find all departments whose name is the same as their building name:
 - $\sigma_{dept_name=building} (department)$



Project Operation

- A unary operation that returns its argument relation, with certain attributes left out.
- Notation:

$$\Pi_{A_1, A_2, A_3, \dots, A_k} (r)$$

where A_1, A_2 are attribute names and r is a relation name.

- The result is defined as the relation of k columns obtained by erasing the columns that are not listed
- Duplicate rows removed from result, since relations are sets



Project Operation (Cont.)

- Example: eliminate the *dept_name* attribute of *instructor*
- Query:

$$\Pi_{ID, name, salary} (instructor)$$

- Result:

<i>ID</i>	<i>name</i>	<i>salary</i>
10101	Srinivasan	65000
12121	Wu	90000
15151	Mozart	40000
22222	Einstein	95000
32343	El Said	60000
33456	Gold	87000
45565	Katz	75000
58583	Califieri	62000
76543	Singh	80000
76766	Crick	72000
83821	Brandt	92000
98345	Kim	80000



Composition of Relational Operations

- The result of a relational-algebra operation is relation and therefore of relational-algebra operations can be composed together into a **relational-algebra expression**.
- Consider the query -- Find the names of all instructors in the Physics department.

$$\Pi_{name}(\sigma_{dept_name = "Physics"}(instructor))$$

- Instead of giving the name of a relation as the argument of the projection operation, we give an expression that evaluates to a relation.

The Dreaded Relax Calculator

- Let's look at an online tool that you will use.
- RelaX (<https://dbis-uibk.github.io/relax/calc/local/uibk/local/0>)
- The calculator:
 - Has an older version of the data from the recommended textbook.
(<https://dbis-uibk.github.io/relax/calc/gist/4f7866c17624ca9dfa85ed2482078be8/relax-silberschatz-english.txt/0>)
 - You can also upload new data.
- Some queries:
 - $\sigma \text{dept_name}='Comp. Sci.' \vee \text{dept_name}='History'$ (department)
 - $\pi \text{name}, \text{dept_name}$ (instructor)
 - $\pi \text{ID}, \text{name}$
 $\sigma \text{dept_name}='Comp. Sci.'$ (instructor)
)

Structured Query Language (I)



History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
 - SQL-86
 - SQL-89
 - SQL-92
 - SQL:1999 (language name became Y2K compliant!)
 - SQL:2003
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
 - Not all examples here may work on your particular system.



SQL Parts

- DML -- provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.
- integrity – the DDL includes commands for specifying integrity constraints.
- View definition -- The DDL includes commands for defining views.
- Transaction control –includes commands for specifying the beginning and ending of transactions.
- Embedded SQL and dynamic SQL -- define how SQL statements can be embedded within general-purpose programming languages.
- Authorization – includes commands for specifying access rights to relations and views.

SQL Language Statements

The core SQL language statements are:

- SELECT: Implements both σ , π
- INSERT
- UPDATE
- DELETE
- CREATE TABLE
- ALTER TABLE
- JOIN, which is an operator within SELECT.
- Many, if not most, SQL statements:
 - Implement multiple relational algebra expressions.
 - Cannot easily (or at all) be represented in relational algebra.

$$\begin{aligned} \pi \text{ ID, name } (\\ & \quad \sigma \text{ dept_name='Comp. Sci.' } (\text{instructor}) \\) \\ = \\ \text{SELECT ID, name FROM instructor} \\ \text{WHERE} \\ \text{dept_name='Comp. Sci.'} \end{aligned}$$



Basic Query Structure

- A typical SQL query has the form:

```
select  $A_1, A_2, \dots, A_n$ 
  from  $r_1, r_2, \dots, r_m$ 
    where  $P$ 
```

- A_i represents an attribute
- R_i represents a relation
- P is a predicate.
- The result of an SQL query is a relation.

Note:

- The SELECT ... FROM ... WHERE ... Combines two relational operators, σ and Π .
- Actually, it also combines other operators, e.g. \times



The select Clause

- The **select** clause lists the attributes desired in the result of a query
 - corresponds to the projection operation of the relational algebra
- Example: find the names of all instructors:

```
select name  
      from instructor
```
- NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)
 - E.g., $Name \equiv NAME \equiv name$
 - Some people use upper case wherever we use bold font.



The select Clause (Cont.)

- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, insert the keyword **distinct** after `select`.
- Find the department names of all instructors, and remove duplicates

```
select distinct dept_name  
from instructor
```

- The keyword **all** specifies that duplicates should not be removed.

```
select all dept_name  
from instructor
```



The select Clause (Cont.)

- An asterisk in the select clause denotes “all attributes”

```
select *
from instructor
```

- An attribute can be a literal with no **from** clause

```
select '437'
```

- Results is a table with one column and a single row with value “437”
- Can give the column a name using:

```
select '437' as FOO
```

- An attribute can be a literal with **from** clause

```
select 'A'
from instructor
```

- Result is a table with one column and N rows (number of tuples in the *instructors* table), each row with value “A”



The select Clause (Cont.)

- The **select** clause can contain arithmetic expressions involving the operation, $+$, $-$, $*$, and $/$, and operating on constants or attributes of tuples.
 - The query:

```
select ID, name, salary/12
from instructor
```

would return a relation that is the same as the *instructor* relation, except that the value of the attribute *salary* is divided by 12.

- Can rename “*salary/12*” using the **as** clause:

```
select ID, name, salary/12 as monthly_salary
```



The where Clause

- The **where** clause specifies conditions that the result must satisfy
 - Corresponds to the selection predicate of the relational algebra.
- To find all instructors in Comp. Sci. dept

```
select name  
from instructor  
where dept_name = 'Comp. Sci.'
```

- SQL allows the use of the logical connectives **and**, **or**, and **not**
- The operands of the logical connectives can be expressions involving the comparison operators <, <=, >, >=, =, and \neq .
- Comparisons can be applied to results of arithmetic expressions
- To find all instructors in Comp. Sci. dept with salary > 80000

```
select name  
from instructor  
where dept_name = 'Comp. Sci.' and salary > 80000
```



The from Clause

- The **from** clause lists the relations involved in the query
 - Corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *instructor X teaches*

```
select *
  from instructor, teaches
```

 - generates every possible instructor – teaches pair, with all attributes from both relations.
 - For common attributes (e.g., *ID*), the attributes in the resulting table are renamed using the relation name (e.g., *instructor.ID*)
- Cartesian product not very useful directly, but useful combined with where-clause condition (selection operation in relational algebra).



Examples

- Find the names of all instructors who have taught some course and the course_id
 - **select** *name, course_id*
from *instructor , teaches*
where *instructor.ID = teaches.ID*
- Find the names of all instructors in the Art department who have taught some course and the course_id
 - **select** *name, course_id*
from *instructor , teaches*
where *instructor.ID = teaches.ID and instructor.dept_name = 'Art'*



Data Definition Language

The SQL data-definition language (DDL) allows the specification of information about relations, including:

- The schema for each relation.
- The type of values associated with each attribute.
- The Integrity constraints
- The set of indices to be maintained for each relation.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.



Create Table Construct

- An SQL relation is defined using the **create table** command:

create table *r*

$(A_1 D_1, A_2 D_2, \dots, A_n D_n,$
 $\text{(integrity-constraint}_1\text{)},$
 $\dots,$
 $\text{(integrity-constraint}_k\text{)})$

- r* is the name of the relation
 - each A_i is an attribute name in the schema of relation *r*
 - D_i is the data type of values in the domain of attribute A_i
- Example:

```
create table instructor (  
    ID          char(5),  
    name        varchar(20),  
    dept_name   varchar(20),  
    salary      numeric(8,2))
```

Some Worked Example

Switch to Demos: Jupyter Notebook and DataGrip

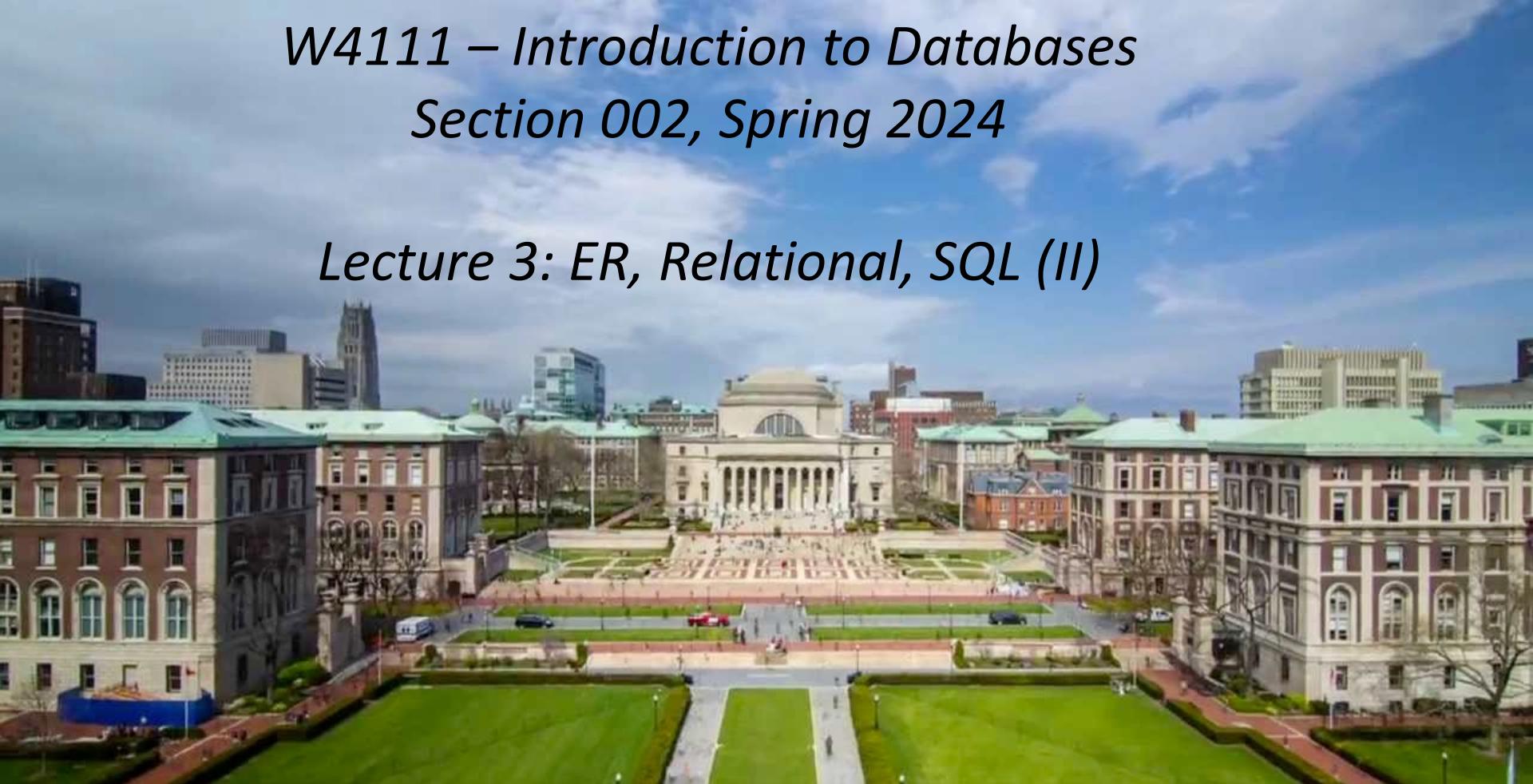
- Drawing a simple data model in Lucidchart.
- Generating sample data in Mockaroo.
- Loading CSV data using DataGrip.
- Editing and modifying schema in SQL.
- Running some queries.
- Doing some relational algebra.

End of Lecture

HW0 and HW1 will come out over the weekend.

*W4111 – Introduction to Databases
Section 002, Spring 2024*

Lecture 3: ER, Relational, SQL (II)



W4111 – Introduction to Databases

Section 002, Spring 2024

Lecture 3: ER, Relational, SQL (II)

We will start in a couple of minutes.

Contents

Contents

- Introduction: Status updates
- ER (Diagram) Modeling – More Complex Scenarios
- The Relational Model and Algebra Continued (Chapter 2)
- SQL Continued (Chapter 3, 4, 5)
- Project Examples:
 - Web Applications and REST
 - Data Engineering and Visualization

Introduction

Status Updates

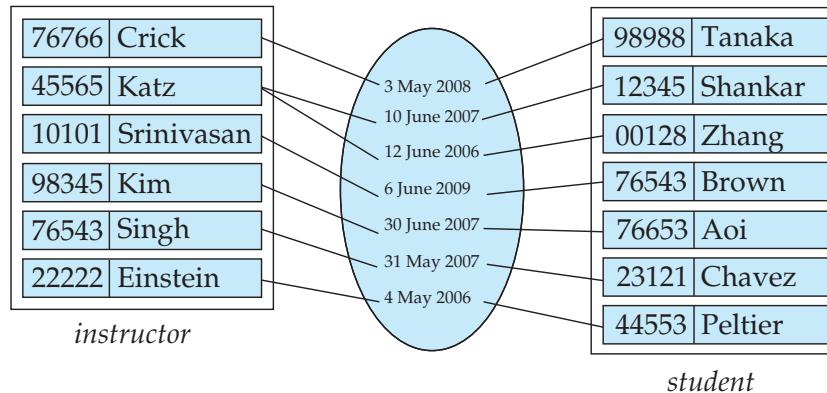
Database design, Entity-Relationship Model (Part 2)

More Complex Relationships



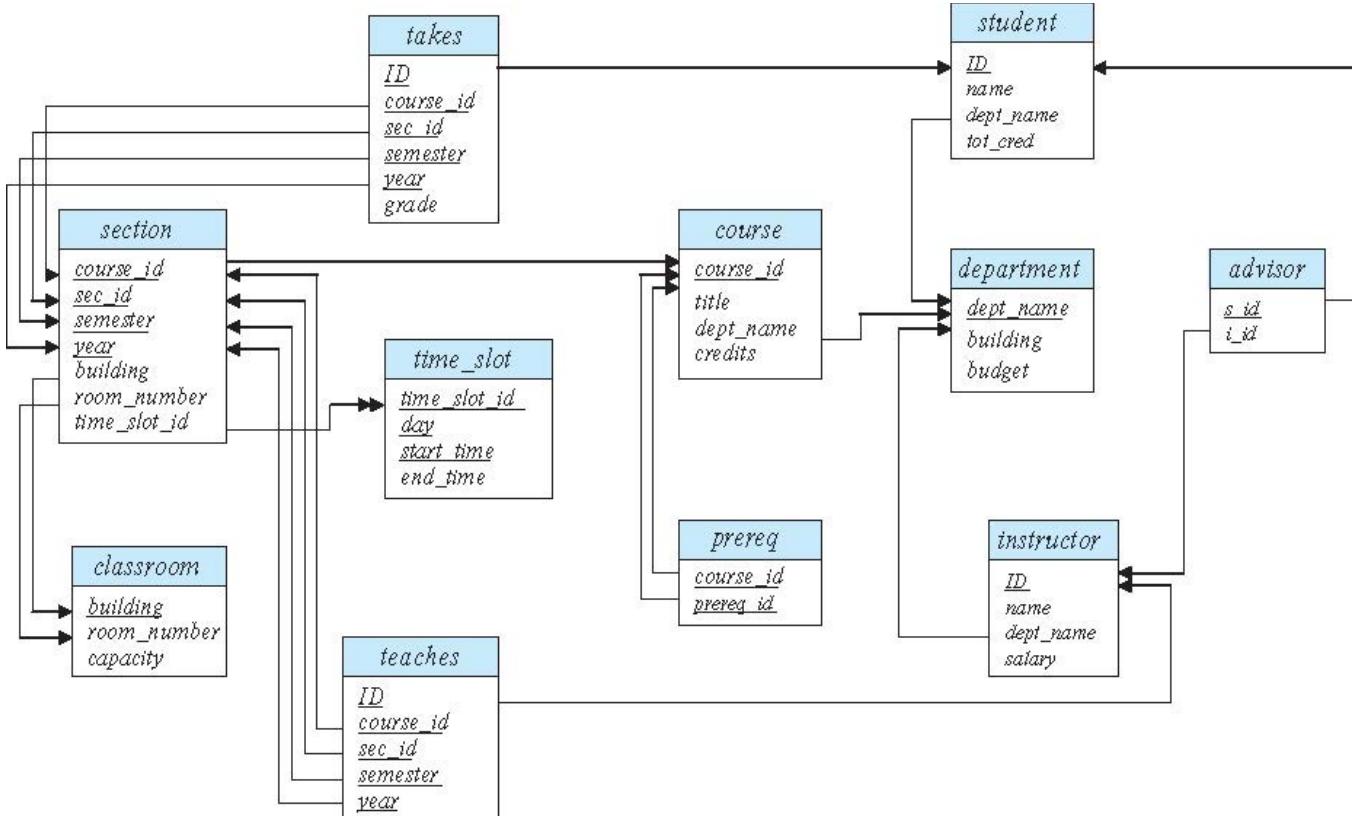
Relationship Sets (Cont.)

- An attribute can also be associated with a relationship set.
- For instance, the *advisor* relationship set between entity sets *instructor* and *student* may have the attribute *date* which tracks when the student started being associated with the advisor



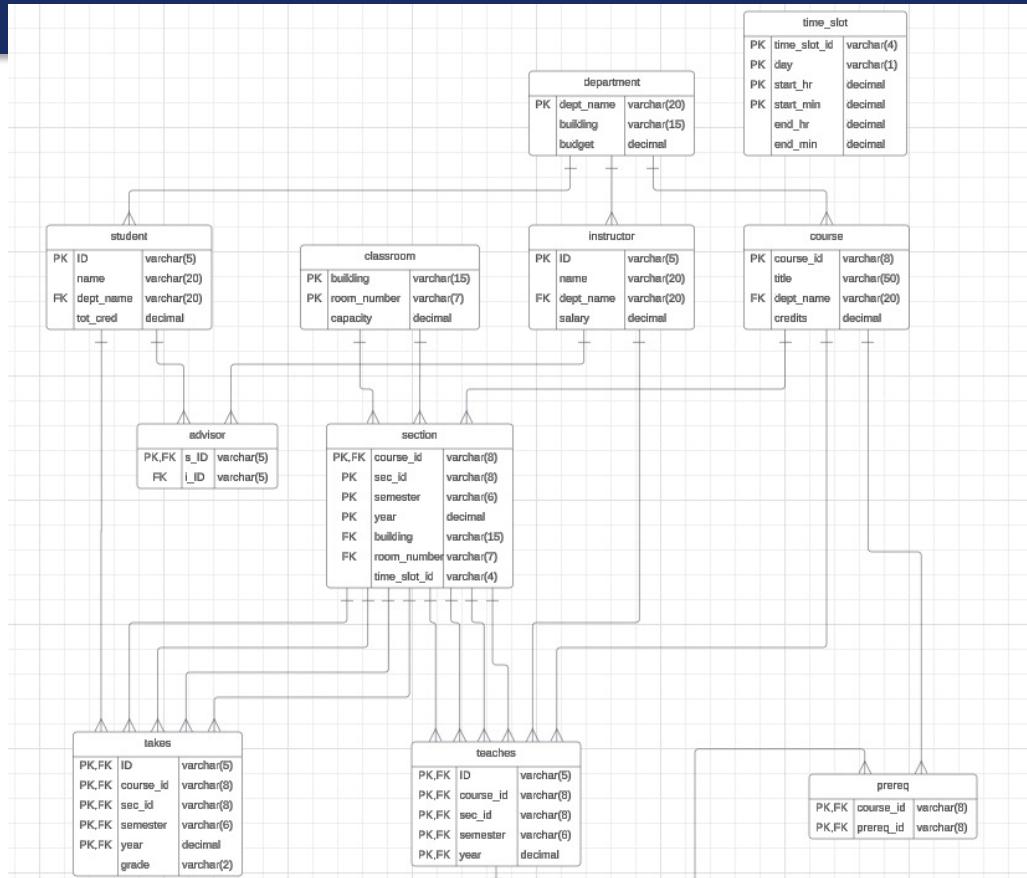


Schema Diagram for University Database



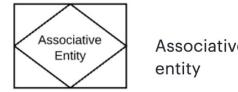
Some Observations

- Reverse engineering seems to have made some mistakes.
- But, notice that sometimes:
 - Relationship are foreign keys, e.g. Section-Classroom
 - Are something else, e.g. Advisor
- This something else is called and Associative Entity.
- SQL requires an Associative Entity if
 - The relationship is many-to-many.
 - There are properties on the relationships.



Associative Entity

- The ER model represents “associations/relationships” as
 - First class “things”
 - That are different from “entities.”
- The SQL model does not have “relationships” or associations as first-class types. You have
 - Tables
 - Columns
 - Keys
 - Constraints
 -
- You can implement some “relationships” using foreign keys. Others require something more complex – an *associative entity*.



Associative entity

Associative entities relate the instances of several entity types. They also contain attributes specific to the relationship between those entity instances.

ERD relationship symbols

Within entity-relationship diagrams, relationships are used to document the interaction between two entities. Relationships are usually verbs such as assign, associate, or track and provide useful information that could not be discerned with just the entity types.

Relationship Symbol	Name	Description
---------------------	------	-------------



Relationship

Relationships are associations between or among entities.

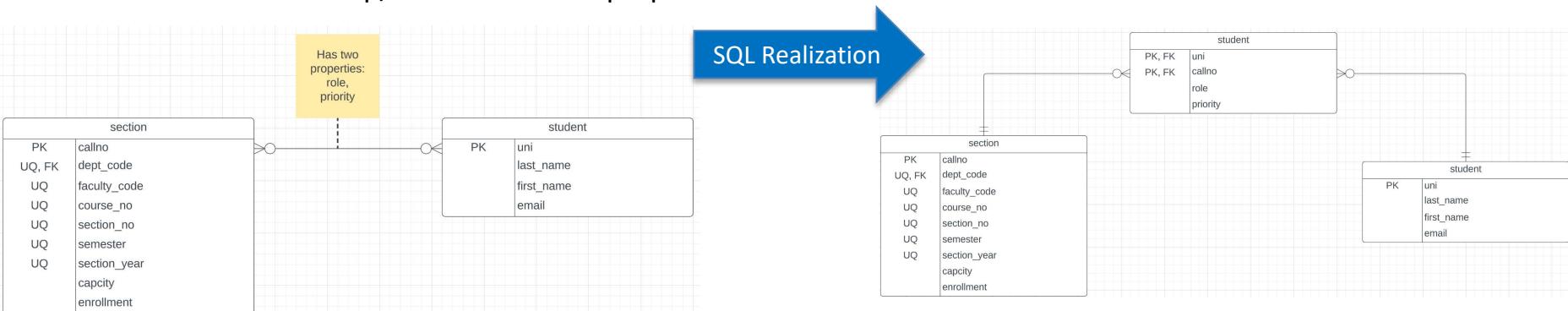


Weak relationship

Weak Relationships are connections between a weak entity and its owner.

Associative Entity

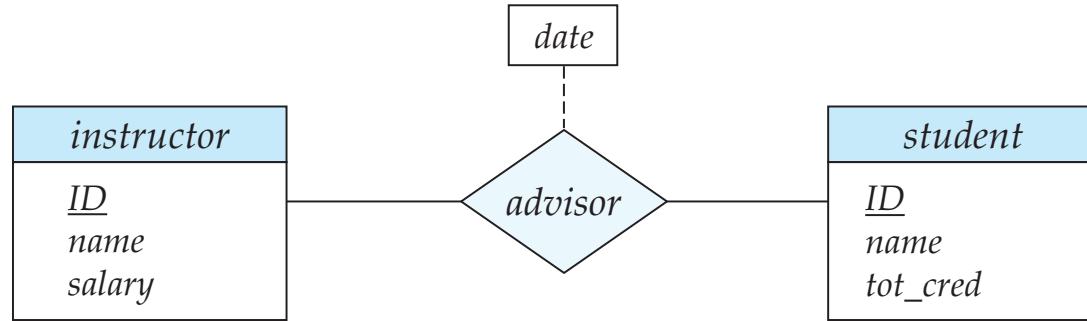
- “An associative entity is a term used in relational and entity–relationship theory. A relational database requires the implementation of a base relation (or base table) to resolve many-to-many relationships. A base relation representing this kind of entity is called, informally, an associative table.” (https://en.wikipedia.org/wiki/Associative_entity)
- Consider *Students – Sections*:
 - This is many-to-many. There is no way to implement in SQL. You see this in the *Advise*s table in the sample database.
 - The “relationship/association” has properties that are not attributes of the connected entities.



Switch to notebook diagram.



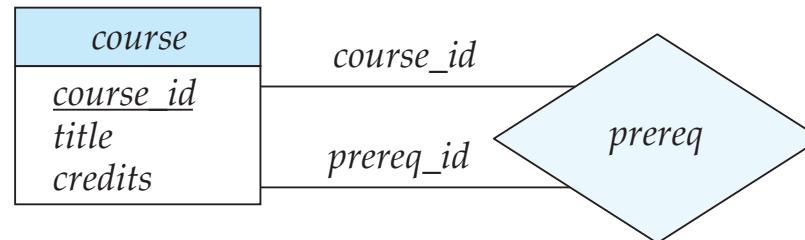
Relationship Sets with Attributes





Roles

- Entity sets of a relationship need not be distinct
 - Each occurrence of an entity set plays a “role” in the relationship
- The labels “*course_id*” and “*prereq_id*” are called **roles**.





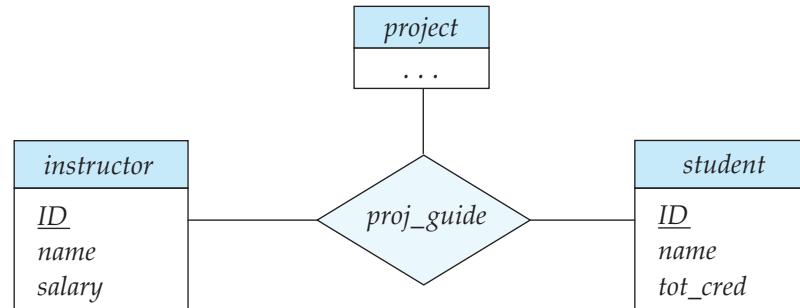
Degree of a Relationship Set

- Binary relationship
 - involve two entity sets (or degree two).
 - most relationship sets in a database system are binary.
- Relationships between more than two entity sets are rare. Most relationships are binary. (More on this later.)
 - Example: *students* work on research *projects* under the guidance of an *instructor*.
 - relationship *proj_guide* is a ternary relationship between *instructor*, *student*, and *project*



Non-binary Relationship Sets

- Most relationship sets are binary
- There are occasions when it is more convenient to represent relationships as non-binary.
- E-R Diagram with a Ternary Relationship





Mapping Cardinality Constraints

- Express the number of entities to which another entity can be associated via a relationship set.
- Most useful in describing binary relationship sets.
- For a binary relationship set the mapping cardinality must be one of the following types:
 - One to one
 - One to many
 - Many to one
 - Many to many

*Apply to –
Game of Thrones*

<https://github.com/jeffreylancaster/game-of-thrones>

~/Dropbox/000/000-Data/GoT

Game of Thrones

- Bottom-Up Data Mapping:
 - “Nouns” usually map to Entity/Entity Set.
 - Nouns inside other nouns often map to:
 - Attribute
 - Relationship
 - Verbs often map to relationships.
 - Adjectives usually map to properties.
- We will start with a subset of the information:
 - Game of Thrones:
 - Episodes
 - Characters
 - IMDB:
 - names_basics
 - title_basics
- Entities Sets
 - Character
 - Season
 - Episode
 - Scene
 - Location, Sublocation
 -
- Relationships
 - Character – Scene
 - Character – Character (e.g. KilledBy)
 - Season – IMDB Title
 - Character – IMDB Name
 -

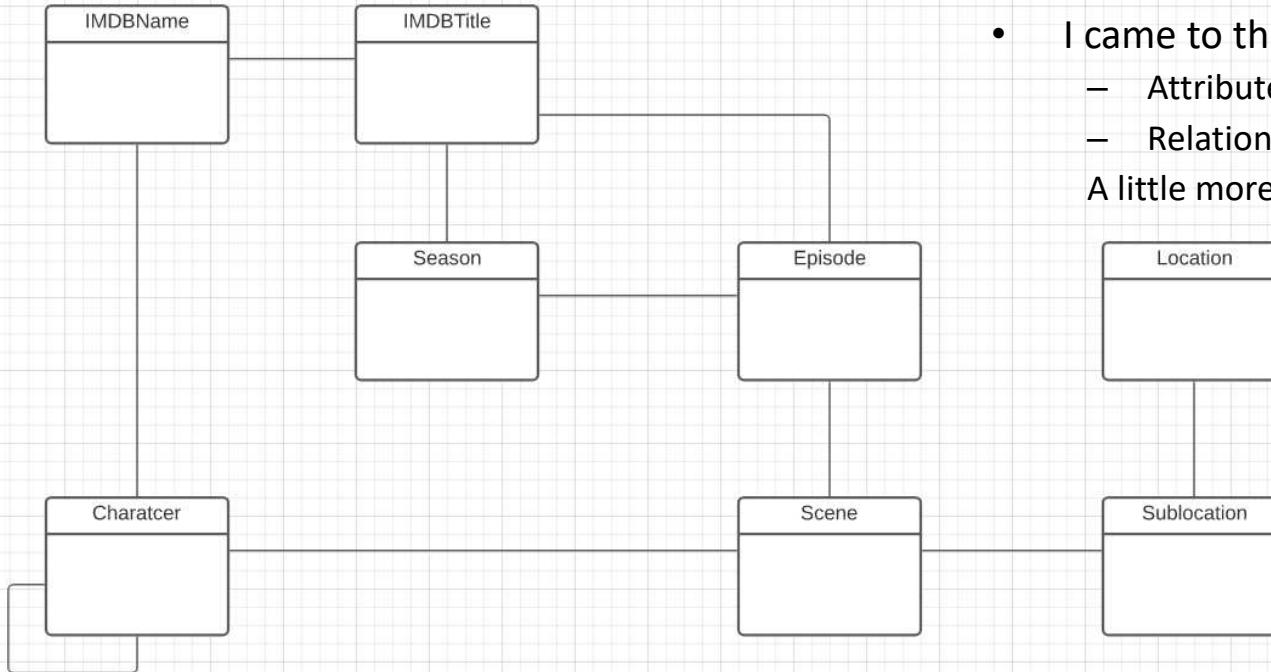
Game of Thrones



Game of Thrones – Conceptual Model

Add shapes in Lucidchart ...

Add Entity Relationship Shapes



- With a little
 - Data exploration
 - Common sense
 - Judgment/experience
- I came to this conceptual model.
 - Attributes unspecified
 - Relationship required/cardinality unspecified.A little more exploration is needed.

Game of Thrones – Conceptual Model

Add!

- Show JSON files and explain.
- Draw diagrams in Lucidchart for
 - GoT: character, season, episode, scene and the relationships.
 - IMDB: namebasic, titles, knownFor corrected, titlePrincipals, etc.
- Loading this data into relational is a bear.
 - GoT:
 - The data is nested document format.
 - Relational is tables.
 - It is doable but requires code and pipelines.
 - I will give you the uncleaned up relational for HW2
 - IMDB:
 - Datasets are very large for a personal/laptop.
 - Many non-atomic columns.
 - [/Users/donaldferguson/Dropbox/000/000-A-Current-Examples/IMDB_GOT_Processing](file:///Users/donaldferguson/Dropbox/000/000-A-Current-Examples/IMDB_GOT_Processing)

Relation Model and Algebra

Assignment

Rename



The Assignment Operation

- It is convenient at times to write a relational-algebra expression by assigning parts of it to temporary relation variables.
- The assignment operation is denoted by \leftarrow and works like assignment in a programming language.
- Example: Find all instructor in the “Physics” and Music department.

$$\text{Physics} \leftarrow \sigma_{\text{dept_name} = \text{“Physics”}}(\text{instructor})$$
$$\text{Music} \leftarrow \sigma_{\text{dept_name} = \text{“Music”}}(\text{instructor})$$
$$\text{Physics} \cup \text{Music}$$

- With the assignment operation, a query can be written as a sequential program consisting of a series of assignments followed by an expression whose value is displayed as the result of the query.

Other Operators

Fun with JOIN

What are all those other Symbols?

- τ order by
- γ group by
- \neg negation
- \div set division
- \bowtie natural join, theta-join
- \bowtie_l left outer join
- \bowtie_r right outer join
- \bowtie_f full outer join
- \bowtie_s left semi join
- \bowtie_{rs} right semi join
- \triangleright anti-join
- Some of these are pretty obscure
 - Division
 - Anti-Join
 - Left semi-join
 - Right semi-join
- Most SQL engines do not support them.
 - You can implement them using combinations of JOIN, SELECT, WHERE,
 - But, I cannot every remember using them in applications I have developed.
- Outer JOIN is very useful, but less common. We will cover.
- There are also some “patterns” or “terms”
 - Equijoin
 - Non-equi join
 - Natural join
 - Theta join
 -
- I may ask you to define these terms on some exams because they may be common internships/job interview questions.

Some Terms

- “A NATURAL JOIN is a JOIN operation that creates an implicit join clause for you based on the common columns in the two tables being joined. Common columns are columns that have the same name in both tables.” (<https://docs.oracle.com/javadb/10.8.3.0/ref/rrefsqjnaturaljoin.html>)
- $\bowtie \rightarrow$ Natural Join in relational algebra.
- So, think about it ...
 - I showed you how to produce all possible pairs.
 - I showed you how to produce all naturally matching pairs.
 - Some simple set operations gives the anti-join.



Equivalent Queries

- There is more than one way to write a query in relational algebra.
- Example: Find information about courses taught by instructors in the Physics department with salary greater than 90,000
- Query 1

$$\sigma_{dept_name = "Physics"} \wedge salary > 90,000 (instructor)$$

- Query 2
- $$\sigma_{dept_name = "Physics"} (\sigma_{salary > 90.000} (instructor))$$
- The two queries are not identical; they are, however, equivalent -- they give the same result on any database.



Equivalent Queries

- There is more than one way to write a query in relational algebra.
- Example: Find information about courses taught by instructors in the Physics department
- Query 1

$$\sigma_{dept_name = "Physics"}(instructor \bowtie_{instructor.ID = teaches.ID} teaches)$$

- Query 2
- $$(\sigma_{dept_name = "Physics"}(instructor)) \bowtie_{instructor.ID = teaches.ID} teaches$$
- The two queries are not identical; they are, however, equivalent -- they give the same result on any database.

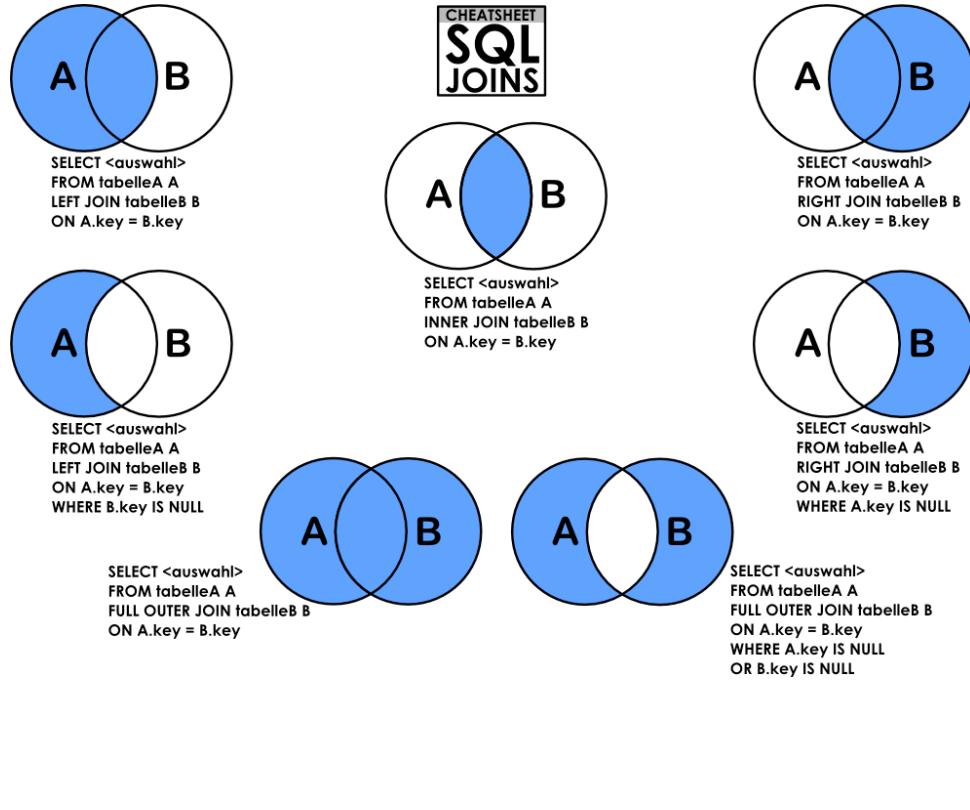
What are all those other Symbols?

- τ order by
- γ group by
- \neg negation
- \div set division
- \bowtie natural join, theta-join
- \bowtie_l left outer join
- \bowtie_r right outer join
- \bowtie_f full outer join
- \bowtie_s left semi join
- \bowtie_r right semi join
- \triangleright anti-join

- Some of the operators are useful and “common,” but not always considered part of the core algebra.
- Some of these are pretty obscure
 - Division
 - Anti-Join
 - Left semi-join
 - Right semi-join
- Most SQL engines do not support them.
 - You can implement them using combinations of JOIN, SELECT, WHERE,
 - But, I cannot every remember using them in applications I have developed.
- Outer JOIN is very useful, but less common. We will cover.
- There are also some “patterns” or “terms”
 - Equijoin
 - Non-equi join
 - Natural join
 - Theta join
 -
- I may ask you to define these terms on some exams or the obscure operators because they may be common internships/job interview questions.

Thinking about JOINS

- Some terms:
 - Natural Join
 - Equality of A and B columns
 - With the same name.
 - Equijoin
 - Explicitly specify columns that must have the same value.
 - $A.x=B.z \text{ AND } A.q=B.w$
 - Theta Join: Arbitrary predicate.
- Inner Join
 - JOIN “matches” rows in A and B.
 - Result contains ONLY the matched pairs.
- What is I want:
 - All the rows that matched.
 - And the rows from A that did not match?
 - OUTER JOIN (\bowtie , $\bowtie\!\!\bowtie$)



Do Some Joins

- prereq is interesting
 - All courses with prereqs.
 - All courses and prereqs even if it does not have a prereq.
 - All courses and ones that are prereqs
 - Courses that do not have prereqs
 - Etc.
- See notebook with examples.

Some Examples

- A course and its prerequisites

```
π course_id←course.course_id,  
      course_title←course.title,  
      prerequisite_id←prereq.prereq_id,  
      prerequisite_title←p.course_id
```

```
(  
  (course ⋈ prereq)  
  ⋈ prereq.prereq_id=p.course_id  
  ρ p(course))
```

- Join course and prereq tables
 - To get the prereq info
 - To get the course title
 - To get the prerequisite title
- Join course and p tables
 - To get the course id
 - To get the prerequisite id
- But,
 - course.course_id is ambiguous
 - Because the table appears twice.
 - So, I have to “alias” the second use.
- Also, note the use of renaming in the project for column names.

Skip in class
But interesting

Some Examples

- Courses-prerequisites and courses without prereqs.

τ prerequisite_id

$(\pi \text{course_id} \leftarrow \text{course.course_id},$
 $\text{course_title} \leftarrow \text{course.title},$

$\text{prerequisite_id} \leftarrow \text{prereq.prereq_id},$

$\text{prerequisite_title} \leftarrow p.\text{course_id}$

(

$(\text{course} \bowtie \text{prereq})$

$\bowtie \text{prereq.prereq_id} = p.\text{course_id}$

$\rho p(\text{course}))$

)

Skip in class
But interesting

Some Examples

- Courses that are not prerequisites

```
τ prereq_course_id  
(  
    π prereq_course_id←prereq.course_id,  
        course_id←course.course_id,  
        course_title←course.title  
    (  
        prereq ⋙ prereq_id=course.course_id course  
    )  
)
```

Skip in class
But interesting

Relational Algebra

- We will do more examples in lectures, HW assignments and exams.
- The language *is interesting but* can be tedious and confusing.
- Useful in some advanced scenarios:
 - Designing query languages for new databases and data models.
 - Understanding how DBMS implement query processing and optimization.
- Also cover because it does come up in some advanced courses, and may come up in job interviews.
- I give take home exams and homework assignments:
 - I cannot remember some of the more obscure operators.
 - You have to tinker with the expressions to get an answer.

SQL

Assignment

Rename



The Rename Operation

- The SQL allows renaming relations and attributes using the **as** clause:

old-name as new-name

- Find the names of all instructors who have a higher salary than some instructor in 'Comp. Sci'.

- **select distinct** *T.name*
from *instructor as T, instructor as S*
where *T.salary > S.salary and S.dept_name = 'Comp. Sci.'*

- Keyword **as** is optional and may be omitted

instructor as T \equiv *instructor T*

NULL

Codd's 12 Rules

Rule 1: Information Rule

The data stored in a database, may it be user data or metadata, must be a value of some table cell. Everything in a database must be stored in a table format.

Rule 2: Guaranteed Access Rule

Every single data element (value) is guaranteed to be accessible logically with a combination of table-name, primary-key (row value), and attribute-name (column value). No other means, such as pointers, can be used to access data.

Rule 3: Systematic Treatment of NULL Values

The NULL values in a database must be given a systematic and uniform treatment. This is a very important rule because a NULL can be interpreted as one the following – data is missing, data is not known, or data is not applicable.

Rule 4: Active Online Catalog

The structure description of the entire database must be stored in an online catalog, known as data dictionary, which can be accessed by authorized users. Users can use the same query language to access the catalog which they use to access the database itself.

Rule 5: Comprehensive Data Sub-Language Rule

A database can only be accessed using a language having linear syntax that supports data definition, data manipulation, and transaction management operations. This language can be used directly or by means of some application. If the database allows access to data without any help of this language, then it is considered as a violation.

Rule 6: View Updating Rule

All the views of a database, which can theoretically be updated, must also be updatable by the system.

Codd's 12 Rules

Rule 7: High-Level Insert, Update, and Delete Rule

A database must support high-level insertion, updation, and deletion. This must not be limited to a single row, that is, it must also support union, intersection and minus operations to yield sets of data records.

Rule 8: Physical Data Independence

The data stored in a database must be independent of the applications that access the database. Any change in the physical structure of a database must not have any impact on how the data is being accessed by external applications.

Rule 9: Logical Data Independence

The logical data in a database must be independent of its user's view (application). Any change in logical data must not affect the applications using it. For example, if two tables are merged or one is split into two different tables, there should be no impact or change on the user application. This is one of the most difficult rules to apply.

Rule 10: Integrity Independence

A database must be independent of the application that uses it. All its integrity constraints can be independently modified without the need of any change in the application. This rule makes a database independent of the front-end application and its interface.

Rule 11: Distribution Independence

The end-user must not be able to see that the data is distributed over various locations. Users should always get the impression that the data is located at one site only. This rule has been regarded as the foundation of distributed database systems.

Rule 12: Non-Subversion Rule

If a system has an interface that provides access to low-level records, then the interface must not be able to subvert the system and bypass security and integrity constraints.

Codd's 12 Rules

Rule 3: Systematic treatment of null values:

- “Null values (distinct from the empty character string or a string of blank characters and distinct from zero or any other number) are supported in fully relational DBMS for representing **missing information** and **inapplicable** information in a systematic way, independent of data type.”
- Sometimes programmers and database designers are tempted to use “special values” to indicate unknown, missing or inapplicable values.
 - String: “”, “NA”, “UNKNOWN”, ...
 - Numbers: -1, 0, -9999
- Indicators can cause confusion because you have to carefully code some SQL statements to the specific, varying choices programmers made.

NULL and Correct Answers

```
In [4]: 1 %%sql describe aaaaS21Examples.null_examples;  
* mysql+pymysql://dbuser:***@localhost  
3 rows affected.
```

```
Out[4]:
```

Field	Type	Null	Key	Default	Extra
name	varchar(32)	NO	PRI	None	
weight	int	YES		None	
net_worth	int	YES		None	

```
In [5]: 1 %%sql select * from aaaaS21Examples.null_examples;  
* mysql+pymysql://dbuser:***@localhost  
4 rows affected.
```

```
Out[5]:
```

name	weight	net_worth
Joe	100	100
Larry	0	0
Pete	None	None
Tim	200	200

Without NULL, to get a correct answer:

- I must understand the domain to determine “unknown” values or know what choice a developer made.
- Explicitly include “where weight != 0” in all statements.
- And this varies from column to column, table to table, schema to schema, etc.

```
In [7]: 1 %%sql select avg(weight) as avg_weight, avg(net_worth) as avg_net_worth  
2      from aaaaS21Examples.null_examples where name in ('Joe', 'Larry', 'Tim')  
* mysql+pymysql://dbuser:***@localhost  
1 rows affected.
```

```
Out[7]:
```

avg_weight	avg_net_worth
100.0000	100.0000

```
In [9]: 1 %%sql select avg(weight) as avg_weight, avg(net_worth) as avg_net_worth  
2      from aaaaS21Examples.null_examples where name in ('Joe', 'Pete', 'Tim')  
* mysql+pymysql://dbuser:***@localhost  
1 rows affected.
```

```
Out[9]:
```

avg_weight	avg_net_worth
150.0000	150.0000



Null Values

- It is possible for tuples to have a null value, denoted by **null**, for some of their attributes
- **null** signifies an **unknown value** or that a **value does not exist**.
- The result of any arithmetic expression involving **null** is **null**
 - Example: $5 + \text{null}$ returns **null**
- The predicate **is null** can be used to check for null values.
 - Example: Find all instructors whose salary is null.

```
select name  
from instructor  
where salary is null
```

- The predicate **is not null** succeeds if the value on which it is applied is not null.

Note:

- **NULL is an extremely important concept.**
- **You will find it hard to understand for a while.**



Null Values (Cont.)

- SQL treats as **unknown** the result of any comparison involving a null value (other than predicates **is null** and **is not null**).
 - Example: $5 < \text{null}$ or $\text{null} \neq \text{null}$ or $\text{null} = \text{null}$
- The predicate in a **where** clause can involve Boolean operations (**and**, **or**, **not**); thus the definitions of the Boolean operations need to be extended to deal with the value **unknown**.
 - **and** : $(\text{true and unknown}) = \text{unknown}$,
 $(\text{false and unknown}) = \text{false}$,
 $(\text{unknown and unknown}) = \text{unknown}$
 - **or**: $(\text{unknown or true}) = \text{true}$,
 $(\text{unknown or false}) = \text{unknown}$
 $(\text{unknown or unknown}) = \text{unknown}$
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*

Practice with Modifications



Modification of the Database

- Deletion of tuples from a given relation.
- Insertion of new tuples into a given relation
- Updating of values in some tuples in a given relation



Deletion

- Delete all instructors

delete from *instructor*

- Delete all instructors from the Finance department

delete from *instructor*
where *dept_name*= 'Finance';

- *Delete all tuples in the instructor relation for those instructors associated with a department located in the Watson building.*

delete from *instructor*
where *dept_name* **in** (**select** *dept_name*
from *department*
where *building* = 'Watson');



Deletion (Cont.)

- Delete all instructors whose salary is less than the average salary of instructors

```
delete from instructor  
where salary < (select avg (salary)  
         from instructor);
```

- Problem: as we delete tuples from *instructor*, the average salary changes
- Solution used in SQL:
 1. First, compute **avg** (*salary*) and find all tuples to delete
 2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)



Insertion

- Add a new tuple to *course*

```
insert into course
values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

- or equivalently

```
insert into course (course_id, title, dept_name, credits)
values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

- Add a new tuple to *student* with *tot_creds* set to null

```
insert into student
values ('3003', 'Green', 'Finance', null);
```



Insertion (Cont.)

- Make each student in the Music department who has earned more than 144 credit hours an instructor in the Music department with a salary of \$18,000.

```
insert into instructor
    select ID, name, dept_name, 18000
        from student
    where dept_name = 'Music' and total_cred > 144;
```

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation.

Otherwise queries like

```
insert into table1 select * from table1
```

would cause problem



Updates

- Give a 5% salary raise to all instructors

```
update instructor  
    set salary = salary * 1.05
```

- Give a 5% salary raise to those instructors who earn less than 70000

```
update instructor  
    set salary = salary * 1.05  
    where salary < 70000;
```

- Give a 5% salary raise to instructors whose salary is less than average

```
update instructor  
    set salary = salary * 1.05  
    where salary < (select avg (salary)  
                    from instructor);
```



Updates (Cont.)

- Increase salaries of instructors whose salary is over \$100,000 by 3%, and all others by a 5%
 - Write two **update** statements:

```
update instructor
  set salary = salary * 1.03
  where salary > 100000;
update instructor
  set salary = salary * 1.05
  where salary <= 100000;
```

- The order is important
- Can be done better using the **case** statement (next slide)

Aggregate Functions



Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value

avg: average value

min: minimum value

max: maximum value

sum: sum of values

count: number of values

Note: Some database implementations have additional aggregate functions.



Aggregate Functions – Group By

- Find the average salary of instructors in each department
 - `select dept_name, avg (salary) as avg_salary
from instructor
group by dept_name;`

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

<i>dept_name</i>	<i>avg_salary</i>
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Another View

Employees

DEPARTMENT_ID	SALARY
10	5500
20	15000
20	7000
30	12000
30	5100
30	4900
30	5800
30	5600
40	7500
40	8000
50	9000
50	8500
50	9500
50	8500
50	10500
50	10000
50	9500

Sum of Salary in Employees table for each department

DEPARTMENT_ID	SUM(SALARY)
10	5500
20	22000
30	33400
40	15500
50	65550

- GROUP BY column list
 - Forms partitions containing multiple rows.
 - All rows in a partition have the same values for the GROUP BY columns.
- The aggregate functions
 - Merge the non-group by attributes, which may differ from row to row.
 - Into a single value for each attribute.
- The result is one row per distinct set of GROUP BY values.
- There may be multiple non-GROUP BY COLUMNS, each with its own aggregate function.
- You can use HAVING in place of WHERE on the GROUP BY result.



Aggregate Functions Examples

- Find the average salary of instructors in the Computer Science department
 - **select avg (salary)
from instructor
where dept_name= 'Comp. Sci.';**
- Find the total number of instructors who teach a course in the Spring 2018 semester
 - **select count (distinct ID)
from teaches
where semester = 'Spring' and year = 2018;**
- Find the number of tuples in the *course* relation
 - **select count (*)
from course;**



Aggregation (Cont.)

- Attributes in **select** clause outside of aggregate functions must appear in **group by** list

- /* erroneous query */
select *dept_name, ID, avg (salary)*
from *instructor*
group by *dept_name*;



Aggregate Functions – Having Clause

- Find the names and average salaries of all departments whose average salary is greater than 42000

```
select dept_name, avg (salary) as avg_salary  
from instructor  
group by dept_name  
having avg (salary) > 42000;
```

- Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

Fun with JOINs



Joined Relations

- **Join operations** take two relations and return as a result another relation.
- A join operation is a Cartesian product which requires that tuples in the two relations match (under some condition). It also specifies the attributes that are present in the result of the join
- The join operations are typically used as subquery expressions in the **from** clause
- Three types of joins:
 - Natural join
 - Inner join
 - Outer join

Notes:

- You will also hear terms like equi-join, non-equi-join, theta join, semi-join,
- I ask for definitions on exams, but you can just look them up.



Natural Join in SQL

- Natural join matches tuples with the same values for all common attributes, and retains only one copy of each common column.
- List the names of instructors along with the course ID of the courses that they taught
 - **select** *name, course_id*
from *students, takes,*
where *student.ID = takes.ID;*
- Same query in SQL with “natural join” construct
 - **select** *name, course_id*
from *student natural join takes;*



Natural Join in SQL (Cont.)

- The **from** clause can have multiple relations combined using natural join:

```
select A1, A2, ... An
from r1 natural join r2 natural join .. natural join rn
where P;
```



Student Relation

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>tot_cred</i>
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120



Takes Relation

<i>ID</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>	<i>grade</i>
00128	CS-101	1	Fall	2017	A
00128	CS-347	1	Fall	2017	A-
12345	CS-101	1	Fall	2017	C
12345	CS-190	2	Spring	2017	A
12345	CS-315	1	Spring	2018	A
12345	CS-347	1	Fall	2017	A
19991	HIS-351	1	Spring	2018	B
23121	FIN-201	1	Spring	2018	C+
44553	PHY-101	1	Fall	2017	B-
45678	CS-101	1	Fall	2017	F
45678	CS-101	1	Spring	2018	B+
45678	CS-319	1	Spring	2018	B
54321	CS-101	1	Fall	2017	A-
54321	CS-190	2	Spring	2017	B+
55739	MU-199	1	Spring	2018	A-
76543	CS-101	1	Fall	2017	A
76543	CS-319	2	Spring	2018	A
76653	EE-181	1	Spring	2017	C
98765	CS-101	1	Fall	2017	C-
98765	CS-315	1	Spring	2018	B
98988	BIO-101	1	Summer	2017	A
98988	BIO-301	1	Summer	2018	<i>null</i>



student natural join takes

ID	name	dept_name	tot_cred	course_id	sec_id	semester	year	grade
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2017	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2017	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2017	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2017	A
12345	Shankar	Comp. Sci.	32	CS-315	1	Spring	2018	A
12345	Shankar	Comp. Sci.	32	CS-347	1	Fall	2017	A
19991	Brandt	History	80	HIS-351	1	Spring	2018	B
23121	Chavez	Finance	110	FIN-201	1	Spring	2018	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2017	B-
45678	Levy	Physics	46	CS-101	1	Fall	2017	F
45678	Levy	Physics	46	CS-101	1	Spring	2018	B+
45678	Levy	Physics	46	CS-319	1	Spring	2018	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2017	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2017	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2018	A-
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2017	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2018	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2017	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2017	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2018	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2017	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2018	null



Dangerous in Natural Join

- Beware of unrelated attributes with same name which get equated incorrectly
- Example -- List the names of students instructors along with the titles of courses that they have taken
 - Correct version

```
select name, title  
from student natural join takes, course  
where takes.course_id = course.course_id;
```

- Incorrect version

```
select name, title  
from student natural join takes natural join course;
```

- This query omits all (student name, course title) pairs where the student takes a course in a department other than the student's own department.
- The correct version (above), correctly outputs such pairs.



Natural Join with Using Clause

- To avoid the danger of equating attributes erroneously, we can use the “**using**” construct that allows us to specify exactly which columns should be equated.
- Query example

```
select name, title  
from (student natural join takes) join course using (course_id)
```



Join Condition

- The **on** condition allows a general predicate over the relations being joined
- This predicate is written like a **where** clause predicate except for the use of the keyword **on**
- Query example

```
select *  
from student join takes on student_ID = takes_ID
```

- The **on** condition above specifies that a tuple from *student* matches a tuple from *takes* if their *ID* values are equal.
- Equivalent to:

```
select *  
from student , takes  
where student_ID = takes_ID
```



Join Condition (Cont.)

- The **on** condition allows a general predicate over the relations being joined.
- This predicate is written like a **where** clause predicate except for the use of the keyword **on**.
- Query example

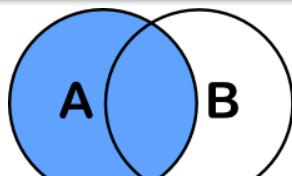
```
select *  
from student join takes on student_ID = takes_ID
```

- The **on** condition above specifies that a tuple from *student* matches a tuple from *takes* if their *ID* values are equal.

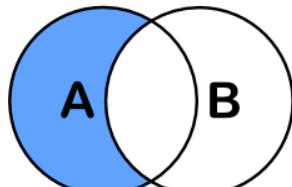
- Equivalent to:

```
select *  
from student, takes  
where student_ID = takes_ID
```

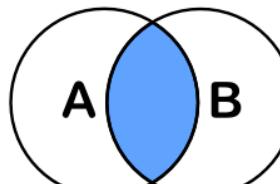
One Way to Think About Joins



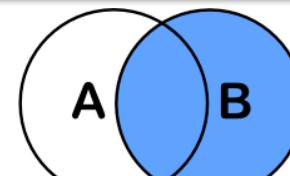
```
SELECT <auswahl>
FROM tabelleA A
LEFT JOIN tabelleB B
ON A.key = B.key
```



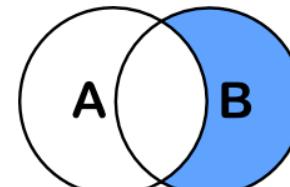
```
SELECT <auswahl>
FROM tabelleA A
LEFT JOIN tabelleB B
ON A.key = B.key
WHERE B.key IS NULL
```



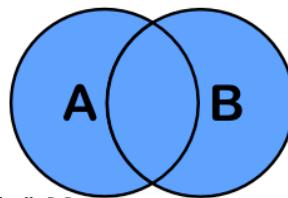
```
SELECT <auswahl>
FROM tabelleA A
INNER JOIN tabelleB B
ON A.key = B.key
```



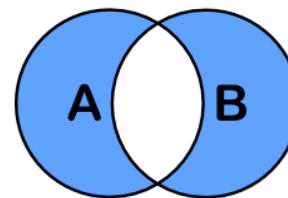
```
SELECT <auswahl>
FROM tabelleA A
RIGHT JOIN tabelleB B
ON A.key = B.key
```



```
SELECT <auswahl>
FROM tabelleA A
RIGHT JOIN tabelleB B
ON A.key = B.key
WHERE A.key IS NULL
```



```
SELECT <auswahl>
FROM tabelleA A
FULL OUTER JOIN tabelleB B
ON A.key = B.key
```



```
SELECT <auswahl>
FROM tabelleA A
FULL OUTER JOIN tabelleB B
ON A.key = B.key
WHERE A.key IS NULL
OR B.key IS NULL
```

Examples in Notebook

Switch to Notebook

Examples
Web Application
Data Engineering Notebook

Sample Projects

Projects

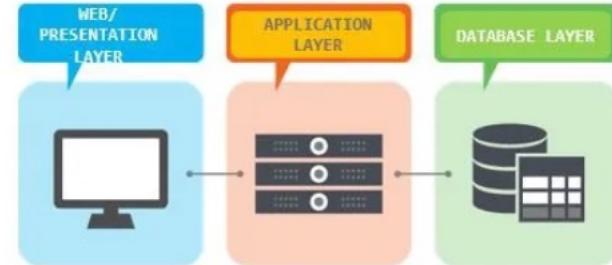
- The programming track will implement a simple, full stack web application.

Full-stack Web Developer

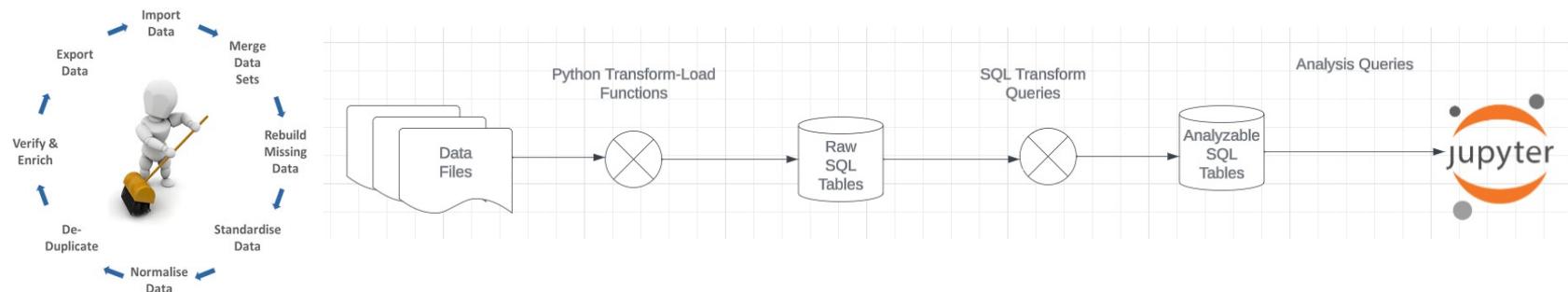
A full-stack web developer is a person who can develop both **client** and **server** software.

In addition to mastering HTML and CSS, he/she also knows how to:

- Program a **browser** (e.g. using JavaScript, jQuery, Angular, or Vue)
- Program a **server** (e.g. using PHP, ASP, Python, or Node)
- Program a **database** (e.g. using SQL, SQLite, or MongoDB)



- The non-programming track will implement a data engineering and visualization process in a Jupyter notebook.



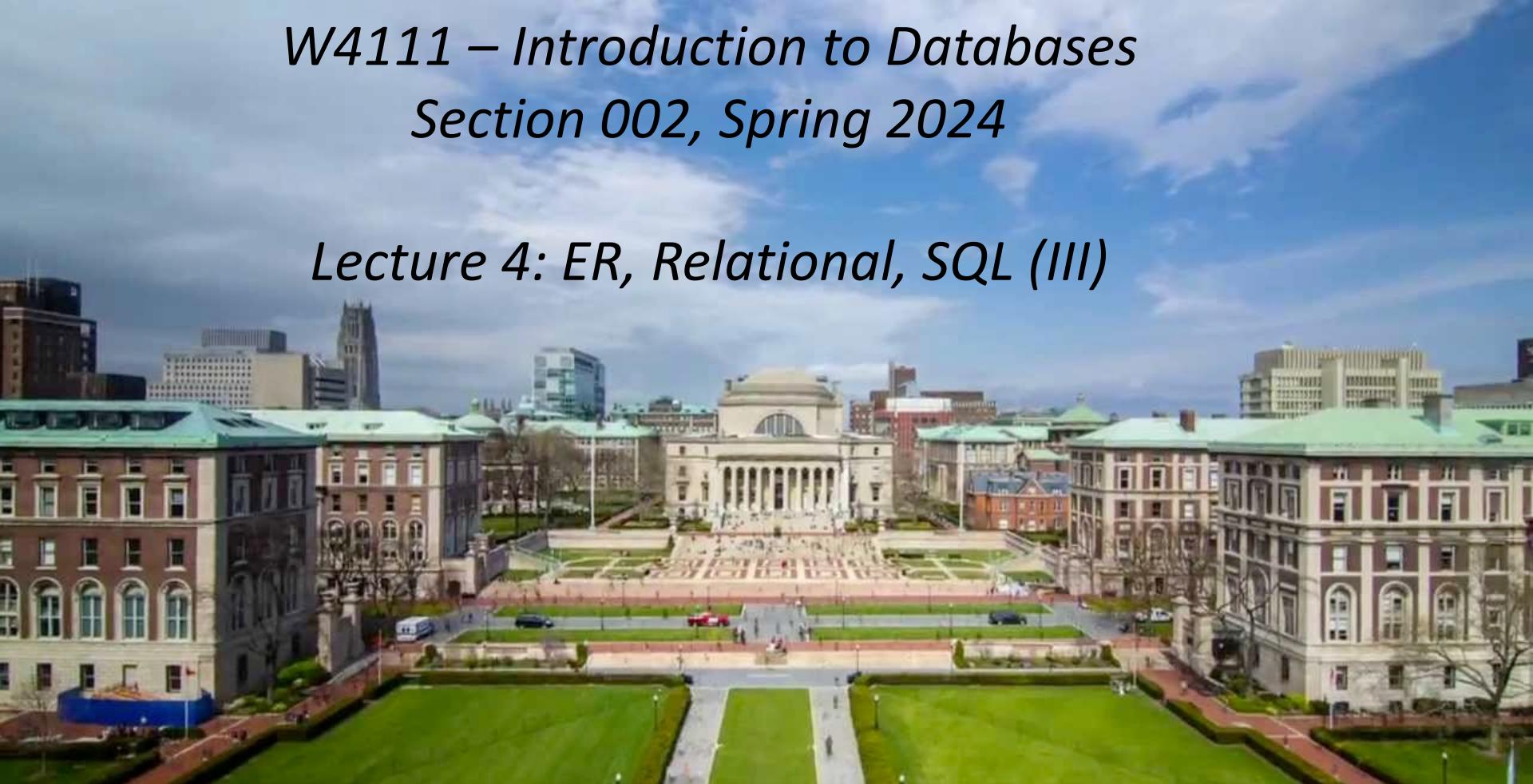
Sample Projects

- Walkthrough of Web Apps:
 - /Users/donaldferguson/Dropbox/000/000-A-Current-Examples/W4111-FastAPI-IMDB-Solution
 - /Users/donaldferguson/Dropbox/000/000-A-Current-Examples/current-dashboard
- Data Engineering:
 - /Users/donaldferguson/Dropbox/000/000-Columbia/2024-Spring/W4111-Intro-to-Databases-Spring-2024/Lectures/s-2024-Lecture-3/More-Data-Engineering.ipynb
 - And others.

The End for Today

*W4111 – Introduction to Databases
Section 002, Spring 2024*

Lecture 4: ER, Relational, SQL (III)



W4111 – Introduction to Databases

Section 002, Spring 2024

Lecture 4: ER, Relational, SQL (III)

We will start in a couple of minutes.

Contents

Contents

- Some Observations
- ER (Diagram) Modeling – More Complex Scenarios: Complex Attributes
- The Relational Model and Algebra Continued (Chapter 3)
- SQL Continued (Chapter 3, 4, 5)
- Project Examples:
 - Web Applications and REST
 - Data Engineering and Visualization

Some Observations

“Create a Table” – Probable not the Best Way to Phrase but Accurate

- Like the relational algebra, SQL is closed under non-update operators, e.g.
 - SELECT
 - UNIONThe result is a table.
- There are a few types of table:
 - Base table
 - Derived table
 - Temporary table
 - View
- SELECT, UNION ... produce a derived table.

SQL 1 and 2 Tables #152



Anonymous

14 hours ago in Assignments - HW1

PIN

STAR

WATCH

63
VIEWS



For SQL1 and SQL2 do we actually have to create the new tables (i.e. CREATE TABLE <xxxx> AS SELECT....) or is just SELECT.... enough?

Comment Edit Delete Endorse ...

1 Answer



Joshua Zhou STAFF

13 hours ago



Just select



Comment Edit Delete Endorse ...

Add comment

- CREATE TEMPORARY TABLE creates a table that exists for the session duration (login duration).
- We will cover *views* later.

Database design, Entity-Relationship Model (Continued)

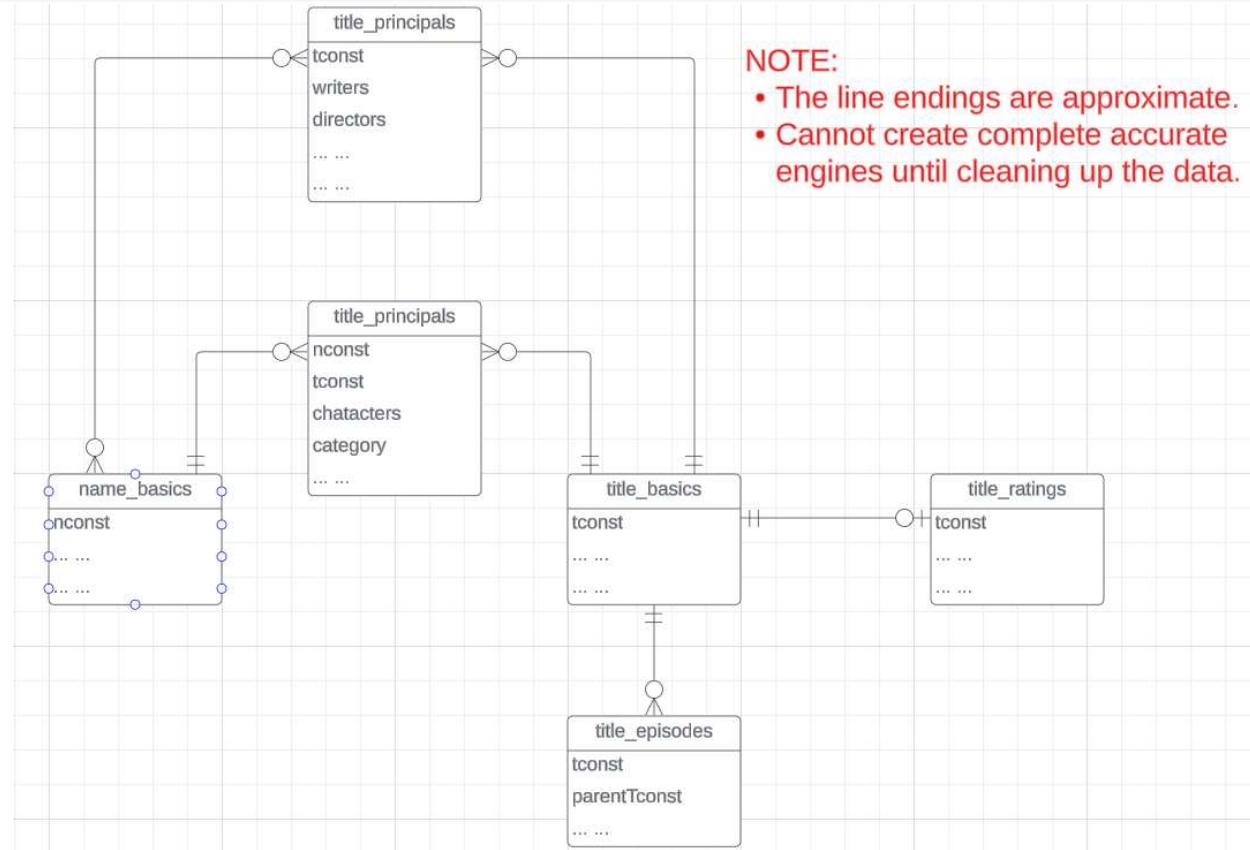
Complex Attributes

Entity Attributes

- The relational model and well-designed SQL schema have *atomic attributes*.
- There are several ways to think about attributes:
 - Simple vs Composite
 - Single Valued versus Multi-valued
 - Derived or Not Derived
- And, there can be combinations, for example a Composite, Multi-Value Attribute. Phone number is an example:
 - A phone number is a composite (+1, 914-555-1212)
 - A customer may have several: work, home, mobile,
- Examining *name_basics* from the IMDB dataset is interesting.

IMDB Free Data

- [IMDB Free Dataset](#) is a set of TSV files:
 - Title Basics
 - Title AKAS
 - Title Crew
 - Title Episodes
 - Title Principals
 - Title Ratings
 - Name Basics
- The data needs a lot of “clean up.”



Example from IMDB

Switch to notebook

- Consider name_basics

	nconst	primaryName	birthYear	deathYear	primaryProfession	knownForTitles
1	nm0000001	Fred Astaire	1899	1987	soundtrack,actor,miscellaneous	tt0050419,tt0031983,tt0072308,tt0053137
2	nm0000002	Lauren Bacall	1924	2014	actress,soundtrack	tt0071877,tt0117057,tt0037332,tt0038355
3	nm0000003	Brigitte Bardot	1934	<null>	actress,soundtrack,music_department	tt0049189,tt0056404,tt0057345,tt0054452
4	nm0000004	John Belushi	1949	1982	actor,soundtrack,writer	tt0077975,tt0072562,tt0080455,tt0078723
5	nm0000005	Ingmar Bergman	1918	2007	writer,director,actor	tt0050986,tt0060827,tt0069467,tt0050976
6	nm0000006	Ingrid Bergman	1915	1982	actress,soundtrack,producer	tt0077711,tt0038109,tt0034583,tt0036855
7	nm0000007	Humphrey Bogart	1899	1957	actor,soundtrack,producer	tt0043265,tt0034583,tt0042593,tt0037382
8	nm0000008	Marlon Brando	1924	2004	actor,soundtrack,director	tt0078788,tt0068646,tt0070849,tt0047296
9	nm0000009	Richard Burton	1925	1984	actor,soundtrack,producer	tt0061184,tt0087803,tt0057877,tt0059749
10	nm0000010	James Cagney	1899	1986	actor,soundtrack,director	tt0029870,tt0035575,tt0042041,tt0055256

- There
 - Is one composite attribute, primaryName.
 - Are two multivalued attributes: primaryProfession, knownForTitles
 - knownForTitles is also tricky, which we will see.
 - Names are also a little tricky

Generated Attributes

The screenshot shows the Mockaroo interface for generating mock data. It displays four fields being configured:

- dept_code:** Custom List with options: COMS, EENG, ECON, MATH, DSCI. Settings: random, blank: 0%, sum.
- faculty_code:** Custom List with options: E, W, C, B, G. Settings: random, blank: 0%, sum.
- course_number:** Digit Sequence with pattern: ####. Settings: question mark, blank: 0%, sum.
- course_name:** Words with min: 3, max: 6, blank: 0%. Settings: sum.

Buttons include "+ ADD ANOTHER FIELD" and "GENERATE FIELDS USING AI...". Configuration options at the bottom include "# Rows: 20", "Format: SQL", "Table Name: MOCK_DATA", and "include CREATE TABLE". A blue callout bubble says "Switch to Notebook".

- Generated some mock data.
- Loaded into a make-believe course table in MySQL.

	A	B	C	D	E
1	dept_code	faculty_code	course_number	course_name	
2	MATH	C	3607	non ligula pellentesque ultrices phasellus id	
3	MATH	C	6017	tincidunt lacus at	
4	ECON	B	9385	velit nec nisi vulputate	
5	MATH	B	4605	congue elementum in	
6	ECON	G	2190	lectus in est risus auctor sed	
7	COMS	C	0991	dictumst morbi vestibulum	
8	ECON	C	4109	sed magna at nunc	
9	COMS	C	1845	ipsum ac tellus semper interdum mauris	
10	DSCI	B	4591	id pretium iaculis diam erat	

Relation Model and Algebra

What are all those other Symbols?

- τ order by
- γ group by
- \neg negation
- \div set division
- \bowtie natural join, theta-join
- \bowtie_l left outer join
- \bowtie_r right outer join
- \bowtie_{lf} full outer join
- \bowtie_l left semi join
- \bowtie_r right semi join
- \triangleright anti-join
- Some of these are obscure and truly bizarre
 - Division
 - Anti-Join
 - Left semi-join
 - Right semi-join
- I must look them up again every semester.
- Most SQL engines do not support them.
 - You can implement them using combinations of JOIN, SELECT, WHERE,
 - But, I cannot even remember using them in applications I have developed.
- Outer JOIN is very useful, but less common. We will cover.
- There are also some “patterns” or “terms”
 - Equijoin
 - Non-equi join
 - Natural join
 - Theta join
 -
- I may ask you to define these terms on some exams because they may be common internships/job interview questions.

Semi-Join

Semijoin (\ltimes and \bowtie) [\[edit\]](#)

The left semijoin is a joining similar to the natural join and written as $R \ltimes S$ where R and S are relations.^[b] The result is the set of all tuples in R for which there is a tuple in S that is equal on their common attribute names. The difference from a natural join is that other columns of S do not appear. For example, consider the tables *Employee* and *Dept* and their semijoin:[\[citation needed\]](#)

Employee			Dept		Employee \ltimes Dept		
Name	Empld	DeptName	DeptName	Manager	Name	Empld	DeptName
Harry	3415	Finance	Sales	Sally	Sally	2241	Sales
Sally	2241	Sales	Production	Harriet	Harriet	2202	Production
George	3401	Finance					
Harriet	2202	Production					

More formally the semantics of the semijoin can be defined as follows:

$$R \ltimes S = \{t : t \in R \wedge \exists s \in S(\text{Fun}(t \cup s))\}$$

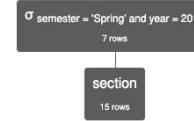
where $\text{Fun}(r)$ is as in the definition of natural join.

The semijoin can be simulated using the natural join as follows. If a_1, \dots, a_n are the attribute names of R , then

$$R \ltimes S = \Pi_{a_1, \dots, a_n} (R \bowtie S).$$

Semi-Join

- $\text{course} \ltimes (\sigma_{\text{semester}=\text{'Spring'}} \wedge \text{year}=2010) (\text{section})$
 - The common column name is *course_id*.
 - The query produces the rows from course for which there is section of the course in the Spring of 2010
- Note:
 - Walk through this in detail in RelaX.
 - We will see the corresponding SQL later.



$\sigma_{\text{semester} = \text{'Spring'}} \wedge \text{year} = 2010 (\text{section})$
Execution time: 1 ms

section.course_id	section.sec_id	section.semester	section.year	section.building	section.instructor
'CS-101'	1	'Spring'	2010	'Packard'	
'CS-315'	1	'Spring'	2010	'Watson'	
'CS-319'	1	'Spring'	2010	'Watson'	
'CS-319'	2	'Spring'	2010	'Taylor'	
'FIN-201'	1	'Spring'	2010	'Packard'	
'HIS-351'	1	'Spring'	2010	'Painter'	
'MU-199'	1	'Spring'	2010	'Packard'	

Anti-Join

Antijoin (\triangleright) [edit]

The antijoin, written as $R \triangleright S$ where R and S are [relations](#),^[c] is similar to the semijoin, but the result of an antijoin is only those tuples in R for which there is *no* tuple in S that is equal on their common attribute names.[\[citation needed\]](#)

For an example consider the tables *Employee* and *Dept* and their antijoin:

Employee		
Name	Empld	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Production

Dept	
DeptName	Manager
Sales	Sally
Production	Harriet

Employee \triangleright Dept		
Name	Empld	DeptName
Harry	3415	Finance
George	3401	Finance

The antijoin is formally defined as follows:

$$R \triangleright S = \{ t : t \in R \wedge \neg \exists s \in S (\text{Fun}(t \cup s)) \}$$

or

$$R \triangleright S = \{ t : t \in R, \text{there is no tuple } s \text{ of } S \text{ that satisfies } \text{Fun}(t \cup s) \}$$

where $\text{Fun}(t \cup s)$ is as in the definition of natural join.

The antijoin can also be defined as the [complement](#) of the semijoin, as follows:

$$R \triangleright S = R - R \ltimes S \tag{5}$$

Given this, the antijoin is sometimes called the anti-semijoin, and the antijoin operator is sometimes written as semijoin symbol with a bar above it, instead of \triangleright .

Using the sample university dataset,
Instructors that are not advisors.

- Anti-Join: $\text{instructor} \triangleright \text{ID}=\text{i_id advisor}$
- One of many equivalent queries:
$$\pi \text{ ID, name, dept_name, salary}$$

$$(\sigma \text{ i_id=NULL}$$

$$(\text{instructor} \bowtie \text{ID}=\text{i_id advisor}))$$

Set Division

- Scroll through https://en.wikipedia.org/wiki/Relational_algebra
- I am not even going to try to figure this out right now.
I already have a headache.

SQL

NULL

Codd's 12 Rules

Rule 1: Information Rule

The data stored in a database, may it be user data or metadata, must be a value of some table cell. Everything in a database must be stored in a table format.

Rule 2: Guaranteed Access Rule

Every single data element (value) is guaranteed to be accessible logically with a combination of table-name, primary-key (row value), and attribute-name (column value). No other means, such as pointers, can be used to access data.

Rule 3: Systematic Treatment of NULL Values

The NULL values in a database must be given a systematic and uniform treatment. This is a very important rule because a NULL can be interpreted as one the following – data is missing, data is not known, or data is not applicable.

Rule 4: Active Online Catalog

The structure description of the entire database must be stored in an online catalog, known as data dictionary, which can be accessed by authorized users. Users can use the same query language to access the catalog which they use to access the database itself.

Rule 5: Comprehensive Data Sub-Language Rule

A database can only be accessed using a language having linear syntax that supports data definition, data manipulation, and transaction management operations. This language can be used directly or by means of some application. If the database allows access to data without any help of this language, then it is considered as a violation.

Rule 6: View Updating Rule

All the views of a database, which can theoretically be updated, must also be updatable by the system.

Codd's 12 Rules

Rule 7: High-Level Insert, Update, and Delete Rule

A database must support high-level insertion, updation, and deletion. This must not be limited to a single row, that is, it must also support union, intersection and minus operations to yield sets of data records.

Rule 8: Physical Data Independence

The data stored in a database must be independent of the applications that access the database. Any change in the physical structure of a database must not have any impact on how the data is being accessed by external applications.

Rule 9: Logical Data Independence

The logical data in a database must be independent of its user's view (application). Any change in logical data must not affect the applications using it. For example, if two tables are merged or one is split into two different tables, there should be no impact or change on the user application. This is one of the most difficult rules to apply.

Rule 10: Integrity Independence

A database must be independent of the application that uses it. All its integrity constraints can be independently modified without the need of any change in the application. This rule makes a database independent of the front-end application and its interface.

Rule 11: Distribution Independence

The end-user must not be able to see that the data is distributed over various locations. Users should always get the impression that the data is located at one site only. This rule has been regarded as the foundation of distributed database systems.

Rule 12: Non-Subversion Rule

If a system has an interface that provides access to low-level records, then the interface must not be able to subvert the system and bypass security and integrity constraints.

Codd's 12 Rules

Rule 3: Systematic treatment of null values:

- “Null values (distinct from the empty character string or a string of blank characters and distinct from zero or any other number) are supported in fully relational DBMS for representing **missing information** and **inapplicable** information in a systematic way, independent of data type.”
- Sometimes programmers and database designers are tempted to use “special values” to indicate unknown, missing or inapplicable values.
 - String: “”, “NA”, “UNKNOWN”, ...
 - Numbers: -1, 0, -9999
- Indicators can cause confusion because you have to carefully code some SQL statements to the specific, varying choices programmers made.

NULL and Correct Answers

```
In [4]: 1 %%sql describe aaaaS21Examples.null_examples;  
* mysql+pymysql://dbuser:***@localhost  
3 rows affected.
```

```
Out[4]:  
Field      Type   Null  Key Default Extra  
name      varchar(32) NO    PRI  None  
weight     int     YES   None  
net_worth  int     YES   None
```

```
In [5]: 1 %%sql select * from aaaaS21Examples.null_examples;  
* mysql+pymysql://dbuser:***@localhost  
4 rows affected.
```

```
Out[5]:  
name  weight  net_worth  
Joe    100      100  
Larry   0        0  
Pete   None     None  
Tim    200      200
```

Without NULL, to get a correct answer:

- I must understand the domain to determine “unknown” values or know what choice a developer made.
- Explicitly include “where weight != 0” in all statements.
- And this varies from column to column, table to table, schema to schema, etc.

```
In [7]: 1 %%sql select avg(weight) as avg_weight, avg(net_worth) as avg_net_worth  
          from aaaaS21Examples.null_examples where name in ('Joe', 'Larry', 'Tim')  
* mysql+pymysql://dbuser:***@localhost  
1 rows affected.
```

```
Out[7]: avg_weight  avg_net_worth  
100.0000      100.0000
```

```
In [9]: 1 %%sql select avg(weight) as avg_weight, avg(net_worth) as avg_net_worth  
          from aaaaS21Examples.null_examples where name in ('Joe', 'Pete', 'Tim')  
* mysql+pymysql://dbuser:***@localhost  
1 rows affected.
```

```
Out[9]: avg_weight  avg_net_worth  
150.0000      150.0000
```



Null Values

- It is possible for tuples to have a null value, denoted by **null**, for some of their attributes
- **null** signifies an **unknown value** or that a **value does not exist**.
- The result of any arithmetic expression involving **null** is **null**
 - Example: $5 + \text{null}$ returns **null**
- The predicate **is null** can be used to check for null values.
 - Example: Find all instructors whose salary is null.

```
select name  
from instructor  
where salary is null
```

- The predicate **is not null** succeeds if the value on which it is applied is not null.

Note:

- **NULL is an extremely important concept.**
- **You will find it hard to understand for a while.**



Null Values (Cont.)

- SQL treats as **unknown** the result of any comparison involving a null value (other than predicates **is null** and **is not null**).
 - Example: $5 < \text{null}$ or $\text{null} \neq \text{null}$ or $\text{null} = \text{null}$
- The predicate in a **where** clause can involve Boolean operations (**and**, **or**, **not**); thus the definitions of the Boolean operations need to be extended to deal with the value **unknown**.
 - **and** : $(\text{true and unknown}) = \text{unknown}$,
 $(\text{false and unknown}) = \text{false}$,
 $(\text{unknown and unknown}) = \text{unknown}$
 - **or**: $(\text{unknown or true}) = \text{true}$,
 $(\text{unknown or false}) = \text{unknown}$
 $(\text{unknown or unknown}) = \text{unknown}$
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*

String Operations Between



String Operations

- SQL includes a string-matching operator for comparisons on character strings. The operator **like** uses patterns that are described using two special characters:
 - percent (%). The % character matches any substring.
 - underscore (_). The _ character matches any character.
- Find the names of all instructors whose name includes the substring “dar”.

```
select name  
from instructor  
where name like '%dar%'
```

- Match the string “100%”

```
like '100 \%' escape '\'
```

in that above we use backslash (\) as the escape character.



String Operations (Cont.)

- Patterns are case sensitive.
- Pattern matching examples:
 - 'Intro%' matches any string beginning with “Intro”.
 - '%Comp%' matches any string containing “Comp” as a substring.
 - '_ _ _' matches any string of exactly three characters.
 - '_ _ _ %' matches any string of at least three characters.
- SQL supports a variety of string operations such as
 - concatenation (using “||”)
 - converting from upper to lower case (and vice versa)
 - finding string length, extracting substrings, etc.



Ordering the Display of Tuples

- List in alphabetic order the names of all instructors

```
select distinct name  
from instructor  
order by name
```
- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
 - Example: **order by name desc**
- Can sort on multiple attributes
 - Example: **order by dept_name, name**

Show notebook for order by example.



Where Clause Predicates

- SQL includes a **between** comparison operator
- Example: Find the names of all instructors with salary between \$90,000 and \$100,000 (that is, $\geq \$90,000$ and $\leq \$100,000$)
 - **select** *name*
from *instructor*
where *salary* **between** 90000 **and** 100000
- Tuple comparison
 - **select** *name, course_id*
from *instructor, teaches*
where (*instructor.ID, dept_name*) = (*teaches.ID, 'Biology'*);

Set Operations



Set Operations

- Find courses that ran in Fall 2017 or in Spring 2018

```
(select course_id from section where sem = 'Fall' and year = 2017)
union
(select course_id from section where sem = 'Spring' and year = 2018)
```

- Find courses that ran in Fall 2017 and in Spring 2018

```
(select course_id from section where sem = 'Fall' and year = 2017)
intersect
(select course_id from section where sem = 'Spring' and year = 2018)
```

- Find courses that ran in Fall 2017 but not in Spring 2018

```
(select course_id from section where sem = 'Fall' and year = 2017)
except
(select course_id from section where sem = 'Spring' and year = 2018)
```



Set Operations (Cont.)

- Set operations **union**, **intersect**, and **except**
 - Each of the above operations automatically eliminates duplicates
- To retain all duplicates use the
 - **union all**,
 - **intersect all**
 - **except all**.

Subqueries

Concepts and Examples

(Including Set Membership)



Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries. A **subquery** is a **select-from-where** expression that is nested within another query.
- The nesting can be done in the following SQL query

```
select A1, A2, ..., An
  from r1, r2, ..., rm
 where P
```

as follows:

- **From clause:** r_i can be replaced by any valid subquery
- **Where clause:** P can be replaced with an expression of the form:

$B <\text{operation}> (\text{subquery})$

B is an attribute and $<\text{operation}>$ to be defined later.

- **Select clause:**

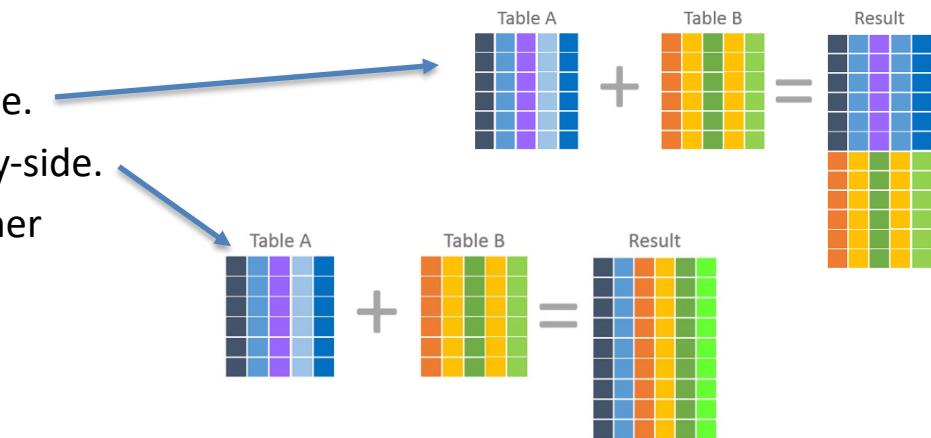
A_i can be replaced by a subquery that generates a single value.

Note:

- This is a little cryptic.
- I think I know what they mean.
- There are some operations we will see later in the material, e.g IN, EXISTS,

Nested Subquery

- The slides that come with the book have surprisingly little material on nested subqueries.
- The concept is:
 - Extremely important.
 - Students often find subqueries more confusing than joins.
 - The relationship/difference of subqueries to joins is often, initial unclear.
- We have seen:
 - Union sort of puts a table on top of a table.
 - Join puts tables sort of puts tables side-by-side.
 - Subquery enables one query to call another during execution like a subfunction.



Consider Some Tables

Takes

ID	course_id	sec_id	semester	year	grade
00128	CS-101	1	Fall	2017	A
00128	CS-347	1	Fall	2017	A-
12345	CS-101	1	Fall	2017	C
12345	CS-190	2	Spring	2017	A
12345	CS-315	1	Spring	2018	A
12345	CS-347	1	Fall	2017	A
19991	HIS-351	1	Spring	2018	B
23121	FIN-201	1	Spring	2018	C+
44553	PHY-101	1	Fall	2017	B-
45678	CS-101	1	Fall	2017	F
45678	CS-101	1	Spring	2018	B+
45678	CS-319	1	Spring	2018	B
54321	CS-101	1	Fall	2017	A-
54321	CS-190	2	Spring	2017	B+
55739	MU-199	1	Spring	2018	A-
76543	CS-101	1	Fall	2017	A
76543	CS-319	2	Spring	2018	A
76653	EE-181	1	Spring	2017	C
98765	CS-101	1	Fall	2017	C-
98765	CS-315	1	Spring	2018	B
98988	BIO-101	1	Summer	2017	A
98988	BIO-301	1	Summer	2018	None

Student

ID	name	dept_name	tot_cred
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

Consider a Subquery Tables

select *, (select name from student where student.id=takes.id) as name from takes;

Takes

ID	course_id	sec_id	semester	year	grade
00128	CS-101	1	Fall	2017	A
00128	CS-347	1	Fall	2017	A-
12345	CS-101	1	Fall	2017	C
12345	CS-190	2	Spring	2017	A
12345	CS-315	1	Spring	2018	A
12345	CS-347	1	Fall	2017	A
19991	HIS-351	1	Spring	2018	B
23121	FIN-201	1	Spring	2018	C+
44553	PHY-101	1	Fall	2017	B-
45678	CS-101	1	Fall	2017	F
45678	CS-101	1	Spring	2018	B+
45678	CS-319	1	Spring	2018	B
54321	CS-101	1	Fall	2017	A-
54321	CS-190	2	Spring	2017	B+
55739	MU-199	1	Spring	2018	A-
76543	CS-101	1	Fall	2017	A
76543	CS-319	2	Spring	2018	A
76653	EE-181	1	Spring	2017	C
98765	CS-101	1	Fall	2017	C-
98765	CS-315	1	Spring	2018	B
98988	BIO-101	1	Summer	2017	A
98988	BIO-301	1	Summer	2018	None

- Assume I wrote a function `find_student_name(x)`
 - Input is an `x`
 - Loops through all students and returns students with `student.ID = x`.
- The query with a subquery above is like:

`result = []`

For `t` in `takes`:

```
new_r = t + find_student_name(t.id)
result.append(new_r)
```

Switch to Notebook

Student

ID	name	dept_name	tot_cred
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120



Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries. A **subquery** is a **select-from-where** expression that is nested within another query.
- The nesting can be done in the following SQL query

```
select A1, A2, ..., An
  from r1, r2, ..., rm
 where P
```

as follows:

- **From clause:** r_i can be replaced by any valid subquery
- **Where clause:** P can be replaced with an expression of the form:

$B <\text{operation}> (\text{subquery})$

B is an attribute and $<\text{operation}>$ to be defined later.

- **Select clause:**
 A_i can be replaced by a subquery that generates a single value.

Note: Subquery MUST return

- A single scalar if in the SELECT.
- A Table if in the FROM.
- If in the WHERE:
 - Either a scalar or a table.
 - Depending on the operation.



Set Membership



Set Membership

- Find courses offered in Fall 2017 and in Spring 2018

```
select distinct course_id
from section
where semester = 'Fall' and year= 2017 and
course_id in (select course_id
from section
where semester = 'Spring' and year= 2018);
```

- Find courses offered in Fall 2017 but not in Spring 2018

```
select distinct course_id
from section
where semester = 'Fall' and year= 2017 and
course_id not in (select course_id
from section
where semester = 'Spring' and year= 2018);
```



Set Membership (Cont.)

- Name all instructors whose name is neither “Mozart” nor Einstein”

```
select distinct name  
from instructor  
where name not in ('Mozart', 'Einstein')
```

- Find the total number of (distinct) students who have taken course sections taught by the instructor with *ID* 10101

```
select count (distinct ID)  
from takes  
where (course_id, sec_id, semester, year) in  
      (select course_id, sec_id, semester, year  
from teaches  
where teaches.ID= 10101);
```

- Note: Above query can be written in a much simpler manner.
The formulation above is simply to illustrate SQL features



Set Comparison



Set Comparison – “some” Clause

- Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department.

```
select distinct T.name  
from instructor as T, instructor as S  
where T.salary > S.salary and S.dept name = 'Biology';
```

- Same query using > **some** clause

```
select name  
from instructor  
where salary > some (select salary  
                      from instructor  
                      where dept name = 'Biology');
```



Definition of “some” Clause

- $F <\text{comp}> \text{some } r \Leftrightarrow \exists t \in r \text{ such that } (F <\text{comp}> t)$
Where comp can be: $<$, \leq , $>$, $=$, \neq

(5 < some

0
5
6

) = true (read: 5 < some tuple in the relation)

(5 < some

0
5

) = false

(5 = some

0
5

) = true

(5 ≠ some

0
5

) = true (since $0 \neq 5$)

$(= \text{some}) \equiv \text{in}$
However, $(\neq \text{some}) \not\equiv \text{not in}$



Set Comparison – “all” Clause

- Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department.

```
select name  
from instructor  
where salary > all (select salary  
                 from instructor  
                 where dept name = 'Biology');
```



Definition of “all” Clause

- $F <\text{comp}> \text{all } r \Leftrightarrow \forall t \in r (F <\text{comp}> t)$

(5 < all

0
5
6

) = false

(5 < all

6
10

) = true

(5 = all

4
5

) = false

(5 ≠ all

4
6

) = true (since $5 \neq 4$ and $5 \neq 6$)

$(\neq \text{all}) \equiv \text{not in}$

However, $(= \text{all}) \not\equiv \text{in}$



Test for Empty Relations

- The **exists** construct returns the value **true** if the argument subquery is nonempty.
- **exists** $r \Leftrightarrow r \neq \emptyset$
- **not exists** $r \Leftrightarrow r = \emptyset$



Use of “exists” Clause

- Yet another way of specifying the query “Find all courses taught in both the Fall 2017 semester and in the Spring 2018 semester”

```
select course_id  
from section as S  
where semester = 'Fall' and year = 2017 and  
exists (select *  
from section as T  
where semester = 'Spring' and year= 2018  
and S.course_id = T.course_id);
```

- **Correlation name** – variable S in the outer query
- **Correlated subquery** – the inner query



Use of “not exists” Clause

- Find all students who have taken all courses offered in the Biology department.

```
select distinct S.ID, S.name
from student as S
where not exists ( (select course_id
                     from course
                     where dept_name = 'Biology')
except
( select T.course_id
  from takes as T
  where S.ID = T.ID));
```

- First nested query lists all courses offered in Biology
- Second nested query lists all courses a particular student took
- Note that $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- Note: Cannot write this query using = all and its variants



Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
- The **unique** construct evaluates to “true” if a given subquery contains no duplicates .
- Find all courses that were offered at most once in 2017

```
select T.course_id
  from course as T
 where unique ( select R.course_id
                  from section as R
                where T.course_id= R.course_id
                  and R.year = 2017);
```



Subqueries in the From Clause



Subqueries in the Form Clause

- SQL allows a subquery expression to be used in the **from** clause
- Find the average instructors' salaries of those departments where the average salary is greater than \$42,000.”

```
select dept_name, avg_salary
  from ( select dept_name, avg (salary) as avg_salary
          from instructor
         group by dept_name)
   where avg_salary > 42000;
```

- Note that we do not need to use the **having** clause
- Another way to write above query

```
select dept_name, avg_salary
  from ( select dept_name, avg (salary)
          from instructor
         group by dept_name)
       as dept_avg (dept_name, avg_salary)
   where avg_salary > 42000;
```



With Clause

- The **with** clause provides a way of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs.
- Find all departments with the maximum budget

```
with max_budget (value) as
  (select max(budget)
   from department)
  select department.name
  from department, max_budget
  where department.budget = max_budget.value;
```



Complex Queries using With Clause

- Find all departments where the total salary is greater than the average of the total salary at all departments

```
with dept_total(dept_name, value) as
  (select dept_name, sum(salary)
   from instructor
   group by dept_name),
dept_total_avg(value) as
  (select avg(value)
   from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value > dept_total_avg.value;
```



Scalar Subquery

- Scalar subquery is one which is used where a single value is expected
- List all departments along with the number of instructors in each department

```
select dept_name,  
       ( select count(*)  
           from instructor  
          where department.dept_name = instructor.dept_name)  
      as num_instructors  
  from department;
```

- Runtime error if subquery returns more than one result tuple

Integrity Constraints, Part 1



Integrity Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
 - A checking account must have a balance greater than \$10,000.00
 - A salary of a bank employee must be at least \$4.00 an hour
 - A customer must have a (non-null) phone number

DFF

- Without integrity constraints in the database, maintaining data correctness requires:
 - Lots of users know what to do and do not make mistakes.
 - Dozens of programs correctly implement constraints in the code and stay up to date on changes.
- Implementing the constraints as part of the schema eliminates many issues.



Constraints on a Single Relation

- **not null**
- **primary key**
- **unique**
- **check (P)**, where P is a predicate



Not Null Constraints

- **not null**
 - Declare *name* and *budget* to be **not null**
name varchar(20) not null
budget numeric(12,2) not null



Unique Constraints

- **unique** (A_1, A_2, \dots, A_m)
 - The unique specification states that the attributes A_1, A_2, \dots, A_m form a candidate key.
 - Candidate keys are permitted to be null (in contrast to primary keys).

Simple Example

- Consider a simple example of an entity class *major*:
 - *major(id, name, track)*
 - “id” is a uniquely generated ID
 - “name” is the major name, e.g. “Computer Science,” “Economics,”
 - “track” is a sub-track/specialty within the major, e.g. “Applications,” “AI/ML,” ...
 - “track” is optional
 - The combination of *(name, track)* is *unique*.
- **Note:** In many DBMS, this automatically creates indices for keys/constraints.
- Switch to Notebook.

```
create table if not exists majors
(
    id          int auto_increment
    primary key,
    major_name  varchar(64) not null,
    major_track varchar(64) null,
    constraint table_name_pk
        unique (major_name, major_track)
);
```



The check clause

- The **check** (P) clause specifies a predicate P that must be satisfied by every tuple in a relation.
- Example: ensure that semester is one of fall, winter, spring or summer

```
create table section
  (course_id varchar (8),
   sec_id varchar (8),
   semester varchar (6),
   year numeric (4,0),
   building varchar (15),
   room_number varchar (7),
   time_slot_id varchar (4),
   primary key (course_id, sec_id, semester, year),
   check (semester in ('Fall', 'Winter', 'Spring', 'Summer')))
```

DFF:

- We could handle the *semester check* with an *enum*.
- Switch to notebook for a slightly different example.



Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
 - Example: If “Biology” is a department name appearing in one of the tuples in the *instructor* relation, then there exists a tuple in the *department* relation for “Biology”.
- Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the primary key of S. A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S.



Referential Integrity (Cont.)

- Foreign keys can be specified as part of the SQL **create table** statement
foreign key (*dept_name*) references *department*
- By default, a foreign key references the primary-key attributes of the referenced table.
- SQL allows a list of attributes of the referenced relation to be specified explicitly.
foreign key (*dept_name*) references *department* (*dept_name*)



Cascading Actions in Referential Integrity

- When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.
- An alternative, in case of delete or update is to cascade

```
create table course (
    ...
    dept_name varchar(20),
    foreign key (dept_name) references department
        on delete cascade
        on update cascade,
    ...
)
```

- Instead of cascade we can use :
 - **set null**,
 - **set default**

DFF:

- I do not like using *cascade*. I think making changes should be explicit.
- Other people disagree.
- You will get some simple practice on HW or exams.

Indexes

Concepts and Examples



Index Creation

- Many queries reference only a small proportion of the records in a table.
- It is inefficient for the system to read every record to find a record with particular value
- An **index** on an attribute of a relation is a data structure that allows the database system to find those tuples in the relation that have a specified value for that attribute efficiently, without scanning through all the tuples of the relation.
- We create an index with the **create index** command

```
create index <name> on <relation-name> (attribute);
```



Index Creation Example

- **create table student**
*(ID varchar (5),
name varchar (20) not null,
dept_name varchar (20),
tot_cred numeric (3,0) default 0,
primary key (ID))*
- **create index studentID_index on student(ID)**
- The query:

```
select *  
from student  
where ID = '12345'
```

can be executed by using the index to find the required record, without looking at all records of *student*

DFF:

- An index on (column1, column2, column3)
- Is also an index on (column1) and (column1, column2)
- But not (column2), (column3), (column2, column3).
- We will see “why” later in the semester.

Sample Projects

Projects

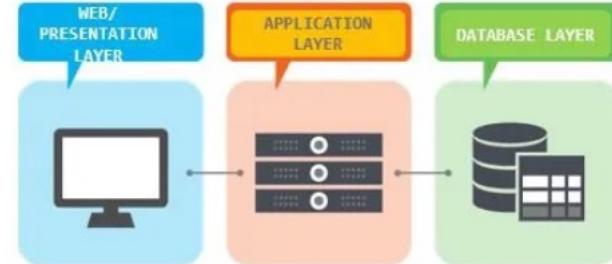
- The programming track will implement a simple, full stack web application.

Full-stack Web Developer

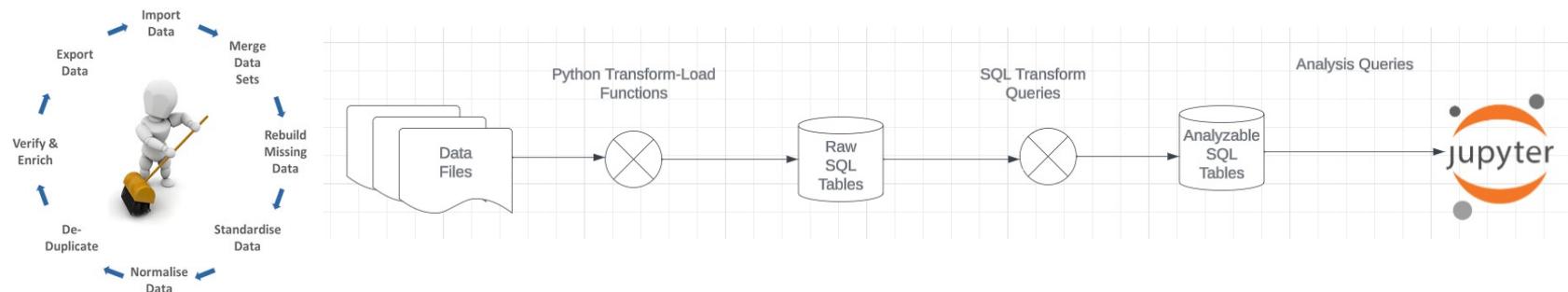
A full-stack web developer is a person who can develop both **client** and **server** software.

In addition to mastering HTML and CSS, he/she also knows how to:

- Program a **browser** (e.g. using JavaScript, jQuery, Angular, or Vue)
- Program a **server** (e.g. using PHP, ASP, Python, or Node)
- Program a **database** (e.g. using SQL, SQLite, or MongoDB)



- The non-programming track will implement a data engineering and visualization process in a Jupyter notebook.



Game of Thrones

- Bottom-Up Data Mapping:
 - “Nouns” usually map to Entity/Entity Set.
 - Nouns inside other nouns often map to:
 - Attribute
 - Relationship
 - Verbs often map to relationships.
 - Adjectives usually map to properties.
- We will start with a subset of the information:
 - Game of Thrones:
 - Episodes
 - Characters
 - IMDB:
 - names_basics
 - title_basics
- Entities Sets
 - Character
 - Season
 - Episode
 - Scene
 - Location, Sublocation
 -
- Relationships
 - Character – Scene
 - Character – Character (e.g. KilledBy)
 - Season – IMDB Title
 - Character – IMDB Name
 -

Game of Thrones

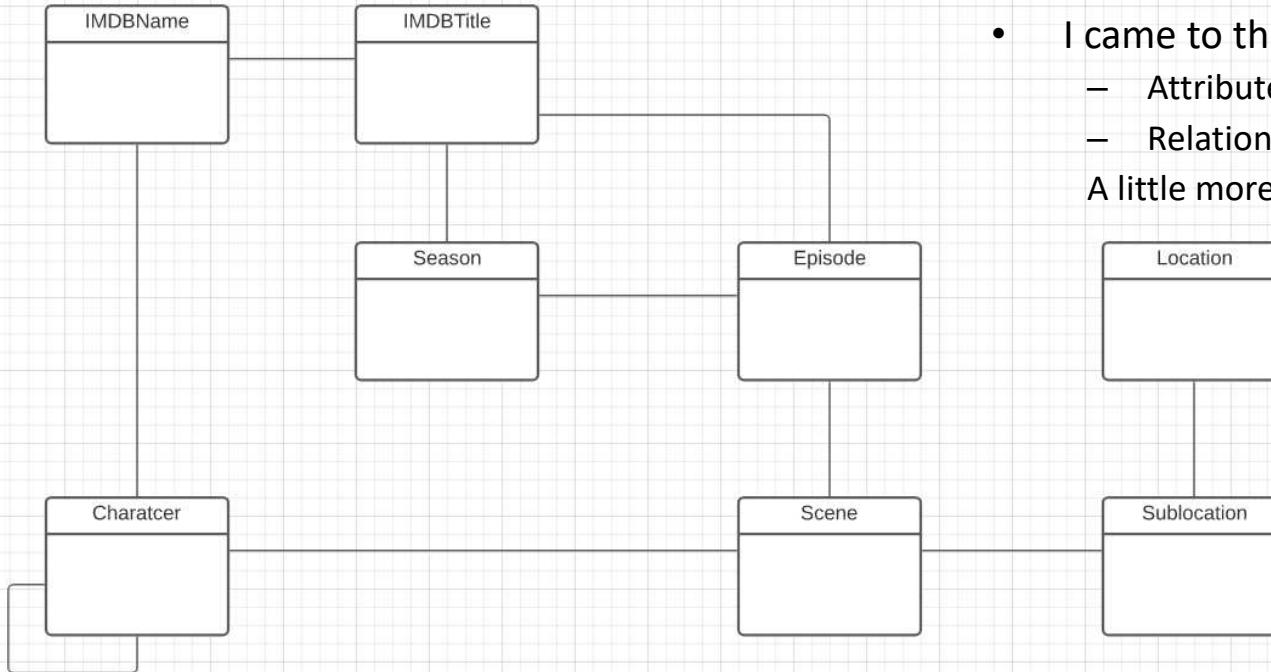
- Bottom-Up Data Mapping:
 - Entities Sets
 - We info
 - IMDB
 - 1. IMDB: <https://developer.imdb.com/non-commercial-datasets/>
 - Do not download.
 - Despite being “tiny” compared to the real world.
 - The datasets are too big for most laptops.
 - 2. Game of Thrones: <https://github.com/jeffreylancaster/game-of-thrones>
 - IMDB Data Structure
 - Character - IMDB Name
 -
 - names_basics
 - title_basics



Game of Thrones – Conceptual Model

Add shapes in Lucidchart ...

Add Entity Relationship Shapes



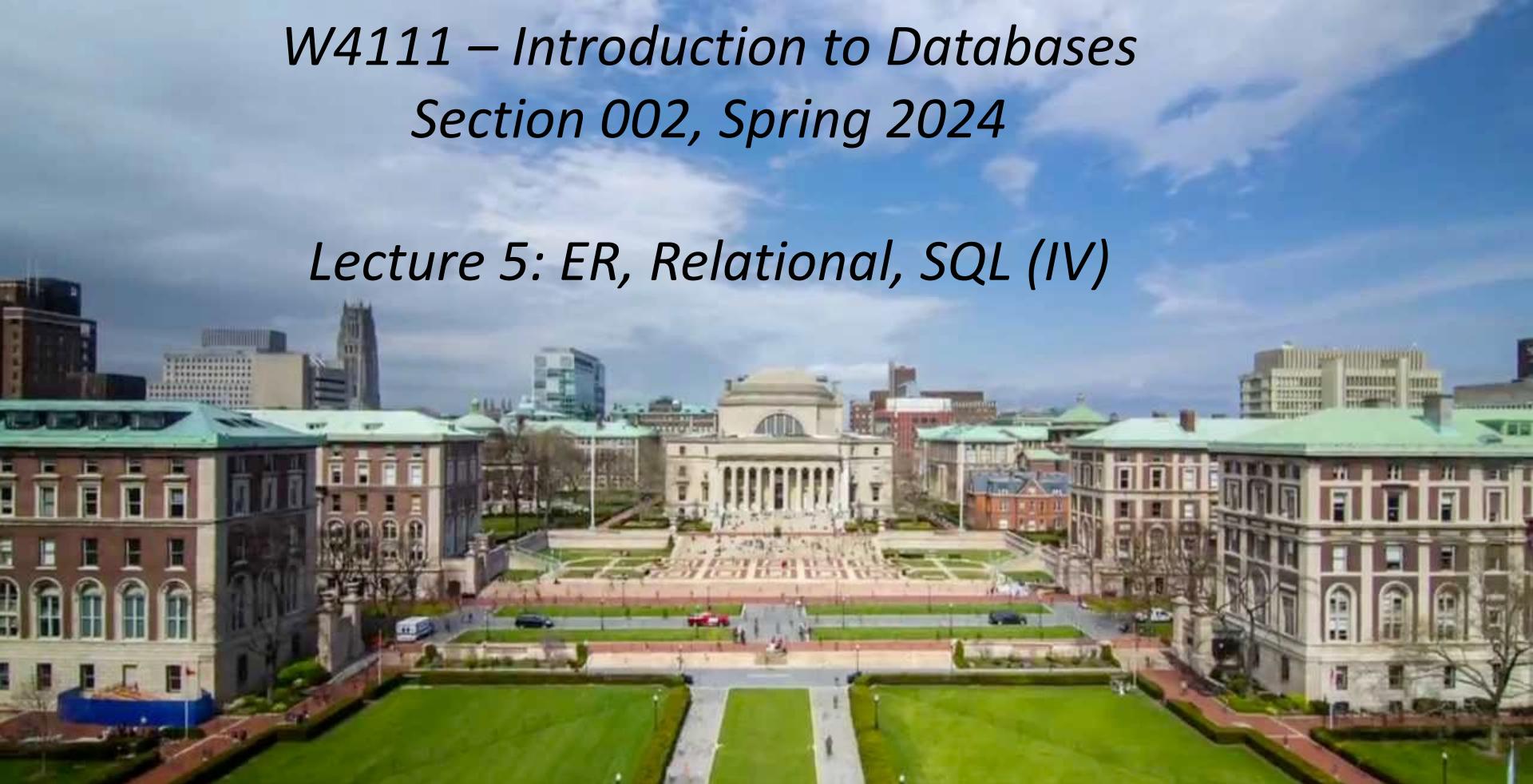
- With a little
 - Data exploration
 - Common sense
 - Judgment/experience
- I came to this conceptual model.
 - Attributes unspecified
 - Relationship required/cardinality unspecified.A little more exploration is needed.

Sample Projects

- Walkthrough of Web Apps:
 - /Users/donaldferguson/Dropbox/000/000-A-Current-Examples/W4111-FastAPI-IMDB-Solution
 - /Users/donaldferguson/Dropbox/000/000-A-Current-Examples/current-dashboard
- Data Engineering:
 - /Users/donaldferguson/Dropbox/000/000-Columbia/2024-Spring/W4111-Intro-to-Databases-Spring-2024/Lectures/s-2024-Lecture-3/More-Data-Engineering.ipynb
 - And others.

*W4111 – Introduction to Databases
Section 002, Spring 2024*

Lecture 5: ER, Relational, SQL (IV)



W4111 – Introduction to Databases

Section 002, Spring 2024

Lecture 5: ER, Relational, SQL (IV)

We will start in a couple of minutes.

Contents

Contents

- Some Observations
- ER (Diagram) Modeling – More Complex Scenarios
- The Relational Model and Algebra Continued (Chapter 3) – Just Kidding
- SQL Continued (Chapter 3, 4, 5)
- Project Examples:
 - Web Applications and REST
 - Data Engineering and Visualization

Some Observations

Database design, Entity-Relationship Model Some Complex Examples



Weak Entity Sets

- Consider a *section* entity, which is uniquely identified by a *course_id*, *semester*, *year*, and *sec_id*.
- Clearly, section entities are related to course entities. Suppose we create a relationship set *sec_course* between entity sets *section* and *course*.
- Note that the information in *sec_course* is redundant, since *section* already has an attribute *course_id*, which identifies the course with which the section is related.
- One option to deal with this redundancy is to get rid of the relationship *sec_course*; however, by doing so the relationship between *section* and *course* becomes implicit in an attribute, which is not desirable.



Weak Entity Sets (Cont.)

- An alternative way to deal with this redundancy is to not store the attribute *course_id* in the *section* entity and to only store the remaining attributes *section_id*, *year*, and *semester*.
 - However, the entity set *section* then does not have enough attributes to identify a particular *section* entity uniquely
- To deal with this problem, we treat the relationship *sec_course* as a special relationship that provides extra information, in this case, the *course_id*, required to identify *section* entities uniquely.
- A **weak entity set** is one whose existence is dependent on another entity, called its **identifying entity**
- Instead of associating a primary key with a weak entity, we use the identifying entity, along with extra attributes called **discriminator** to uniquely identify a weak entity.



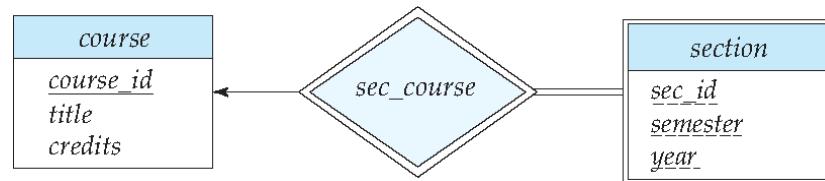
Weak Entity Sets (Cont.)

- An entity set that is not a weak entity set is termed a **strong entity set**.
- Every weak entity must be associated with an identifying entity; that is, the weak entity set is said to be **existence dependent** on the identifying entity set.
- The identifying entity set is said to **own** the weak entity set that it identifies.
- The relationship associating the weak entity set with the identifying entity set is called the **identifying relationship**.
- Note that the relational schema we eventually create from the entity set *section* does have the attribute *course_id*, for reasons that will become clear later, even though we have dropped the attribute *course_id* from the entity set *section*.

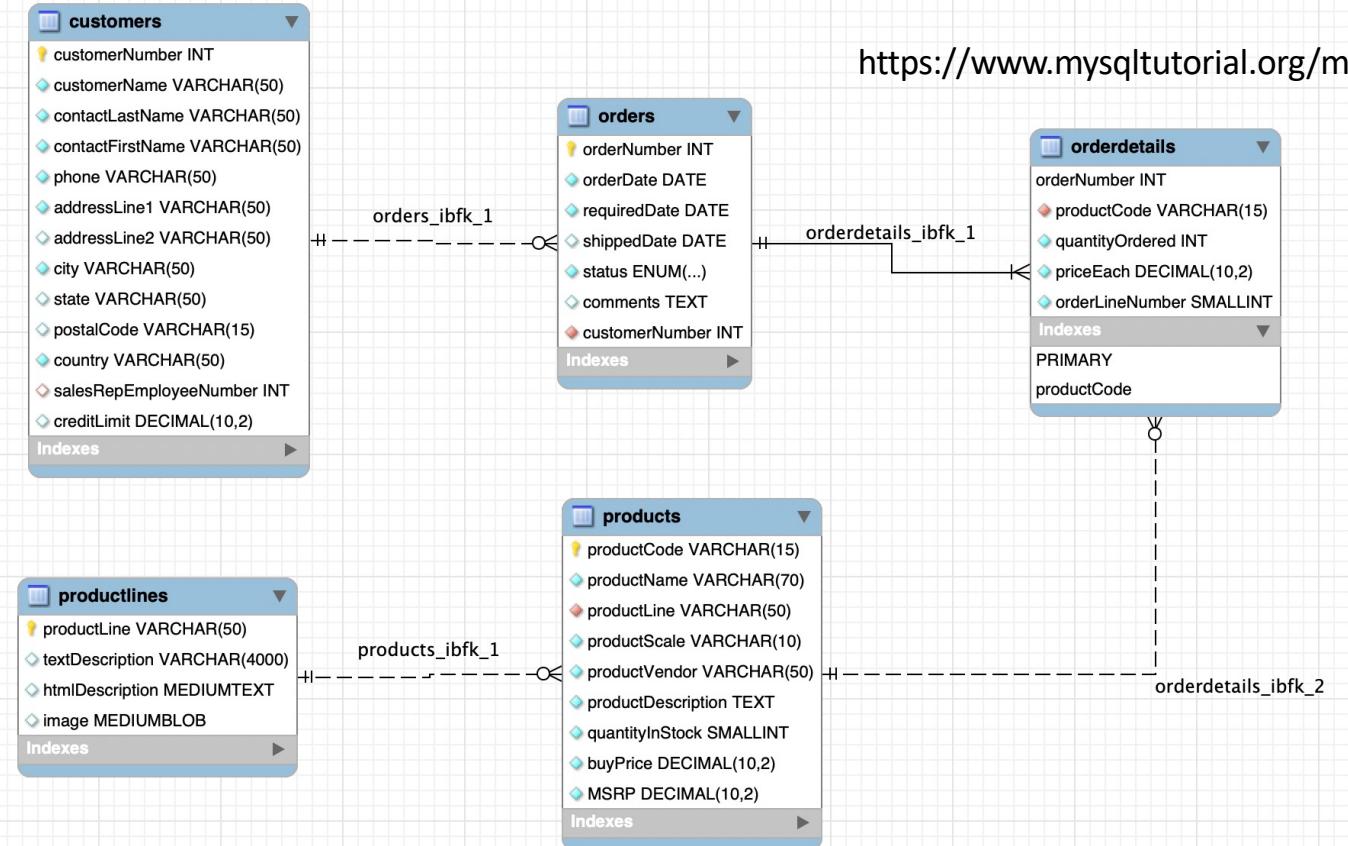


Expressing Weak Entity Sets

- In E-R diagrams, a weak entity set is depicted via a double rectangle.
- We underline the discriminator of a weak entity set with a dashed line.
- The relationship set connecting the weak entity set to the identifying strong entity set is depicted by a double diamond.
- Primary key for *section* – (*course_id*, *sec_id*, *semester*, *year*)



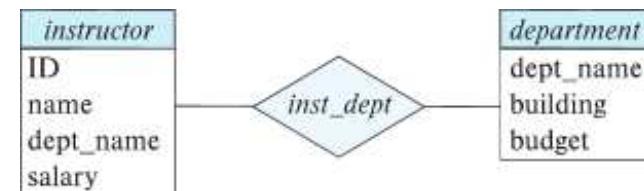
An Example – Classic Models



<https://www.mysqltutorial.org/mysql-sample-database.aspx/>

Redundant Attributes

- Suppose we have entity sets:
 - *instructor*, with attributes: *ID, name, dept_name, salary*
 - *department*, with attributes: *dept_name, building, budget*
- We model the fact that each instructor has an associated department using a relationship set *inst_dept*
- The attribute *dept_name* in *instructor* replicates information present in the relationship and is therefore redundant
 - and needs to be removed.
- BUT: when converting back to tables, in some cases the attribute gets reintroduced, as we will see later.





Design Alternatives

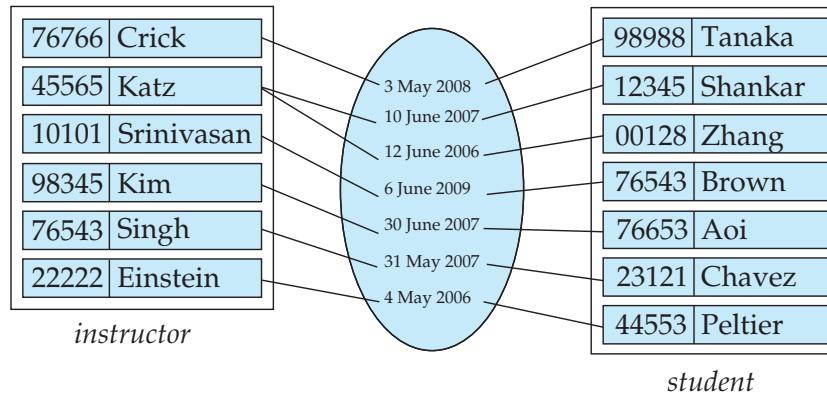
- In designing a database schema, we must ensure that we avoid two major pitfalls:
 - Redundancy: a bad design may result in repeat information.
 - **Redundant representation of information may lead to data inconsistency among the various copies of information**
 - Incompleteness: a bad design may make certain aspects of the enterprise difficult or impossible to model.
- Avoiding bad designs is not enough. There may be a large number of good designs from which we must choose.

**Emphasis
Added**



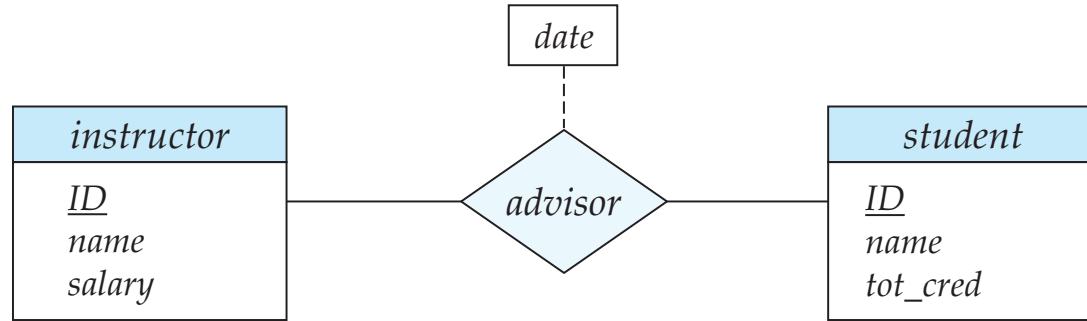
Relationship Sets (Cont.)

- An attribute can also be associated with a relationship set.
- For instance, the *advisor* relationship set between entity sets *instructor* and *student* may have the attribute *date* which tracks when the student started being associated with the advisor





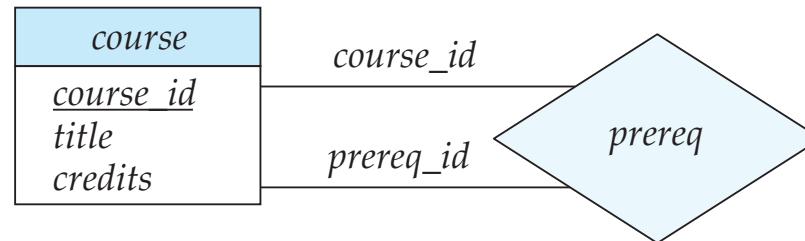
Relationship Sets with Attributes





Roles

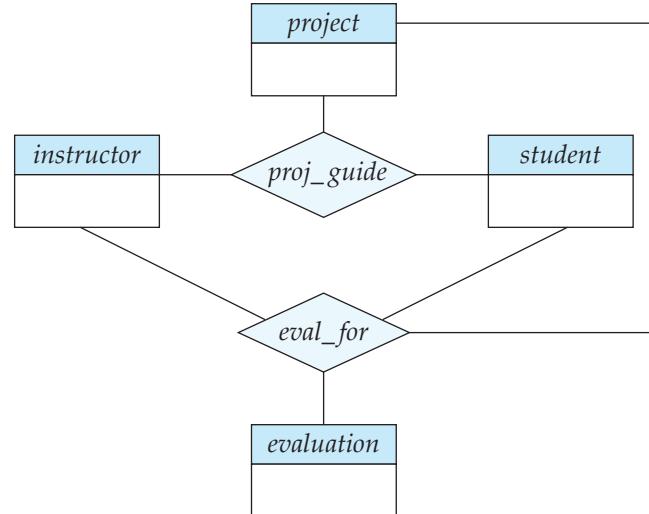
- Entity sets of a relationship need not be distinct
 - Each occurrence of an entity set plays a “role” in the relationship
- The labels “*course_id*” and “*prereq_id*” are called **roles**.





Aggregation

- Consider the ternary relationship *proj_guide*, which we saw earlier
- Suppose we want to record evaluations of a student by a guide on a project





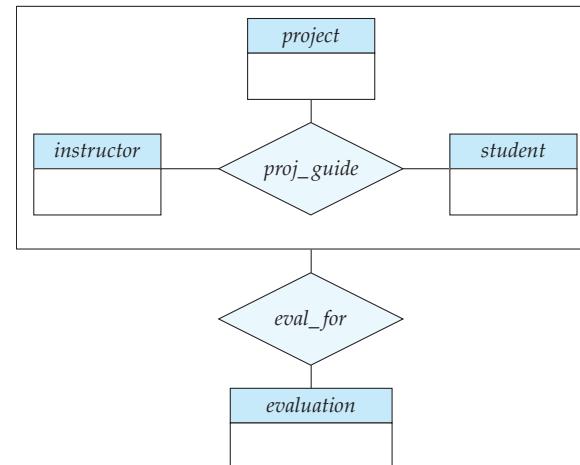
Aggregation (Cont.)

- Relationship sets *eval_for* and *proj_guide* represent overlapping information
 - Every *eval_for* relationship corresponds to a *proj_guide* relationship
 - However, some *proj_guide* relationships may not correspond to any *eval_for* relationships
 - So we can't discard the *proj_guide* relationship
- Eliminate this redundancy via *aggregation*
 - Treat relationship as an abstract entity
 - Allows relationships between relationships
 - Abstraction of relationship into new entity



Aggregation (Cont.)

- Eliminate this redundancy via *aggregation* without introducing redundancy, the following diagram represents:
 - A student is guided by a particular instructor on a particular project
 - A student, instructor, project combination may have an associated evaluation



The simplest way to handle in relational is an associative entity.

Some thoughts here:

<https://www.geeksforgeeks.org/aggregate-data-model-in-nosql/>



Reduction to Relational Schemas

- To represent aggregation, create a schema containing
 - Primary key of the aggregated relationship,
 - The primary key of the associated entity set
 - Any descriptive attributes
- In our example:
 - The schema *eval_for* is:
$$\text{eval_for} (s_ID, project_id, i_ID, evaluation_id)$$
 - The schema *proj_guide* is redundant.



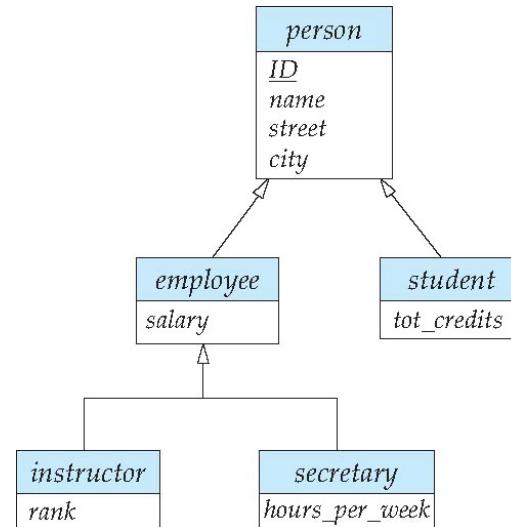
Specialization

- Top-down design process; we designate sub-groupings within an entity set that are distinctive from other entities in the set.
- These sub-groupings become lower-level entity sets that have attributes or participate in relationships that do not apply to the higher-level entity set.
- Depicted by a *triangle* component labeled ISA (e.g., *instructor* “is a” *person*).
- **Attribute inheritance** – a lower-level entity set inherits all the attributes and relationship participation of the higher-level entity set to which it is linked.



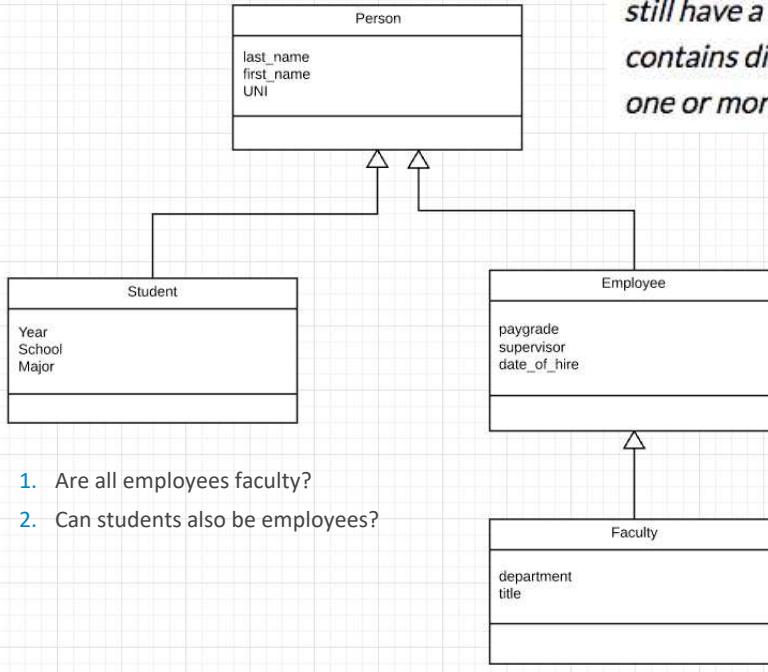
Specialization Example

- **Overlapping** – *employee* and *student*
- **Disjoint** – *instructor* and *secretary*
- Total and partial



Inheritance/Specialization

In the process of designing our entity relationship diagram for a database, we may find that attributes of two or more entities overlap, meaning that these entities seem very similar but still have a few differences. In this case, we may create a subtype of the parent entity that contains distinct attributes. A parent entity becomes a supertype that has a relationship with one or more subtypes.



1. Are all employees faculty?
2. Can students also be employees?

The subclass association line is labeled with specialization constraints. Constraints are described along two dimensions:

1 incomplete/complete

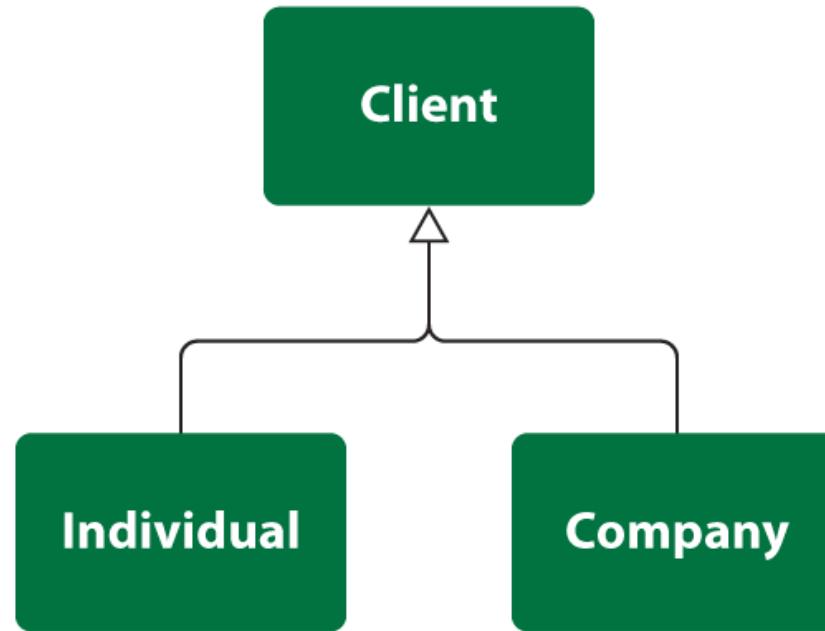
- In an **incomplete** specialization only some instances of the parent class are specialized (have unique attributes). Other instances of the parent class have only the common attributes.
- In a **complete** specialization, every instance of the parent class has one or more unique attributes that are not common to the parent class.

2 disjoint/overlapping

- In a **disjoint** specialization, an object could be a member of only one specialized subclass.
- In an **overlapping** specialization, an object could be a member of more than one specialized subclass.

Specialization

In class Client we distinguish two subtypes: Individual and Company. This specialization is disjoint (client can be an individual or a company) and complete (these are all possible subtypes for supertype).

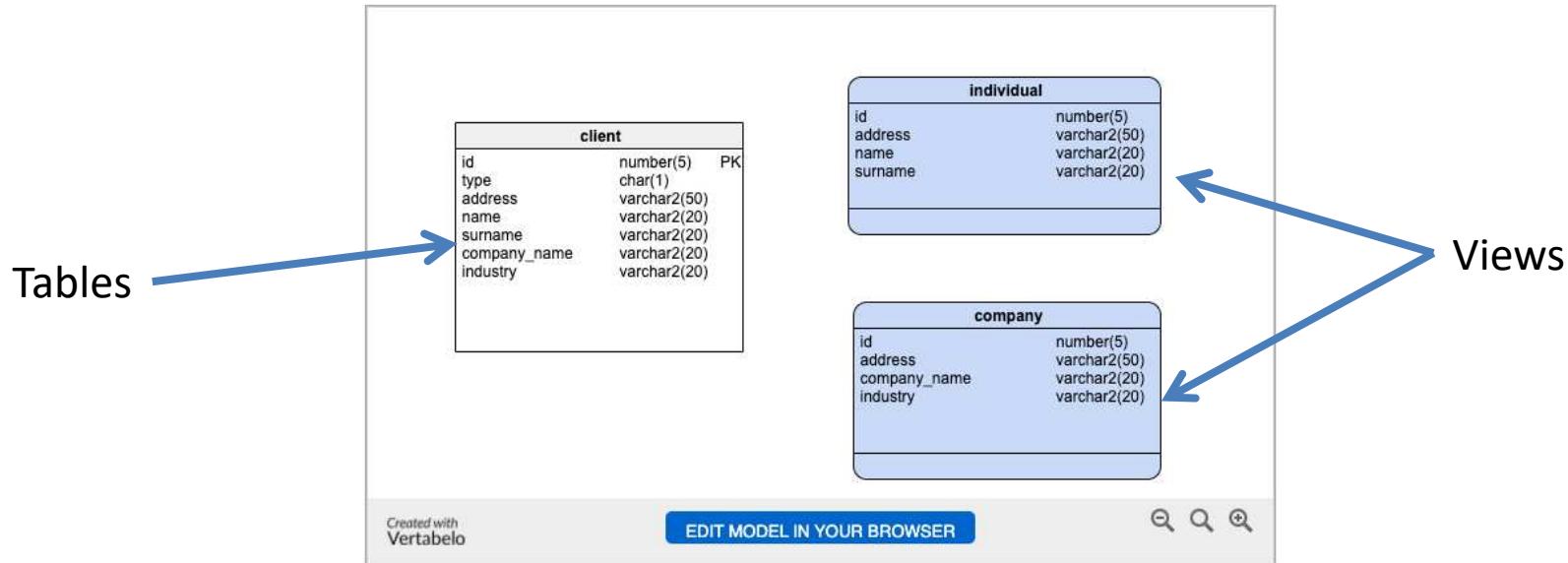


One Table

One table implementation

In a one table implementation, table `client` has attributes of both types.

The diagram below shows the table `client` and two views: `individual` and `company`:

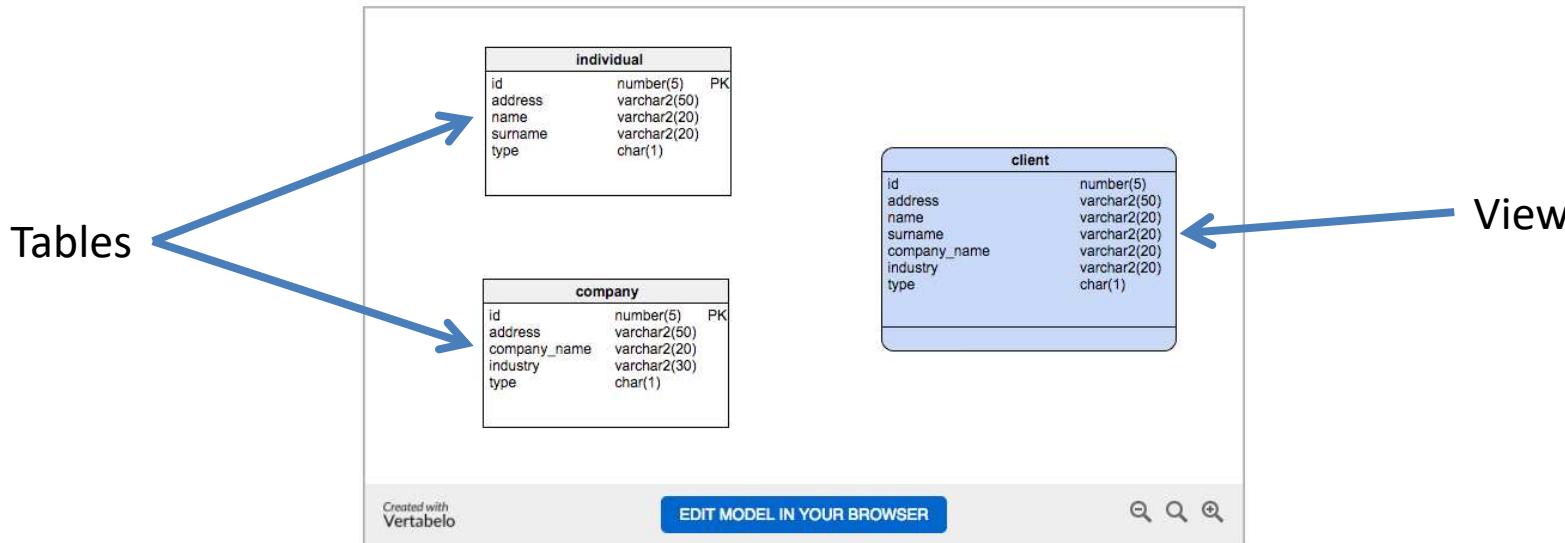


Two Table

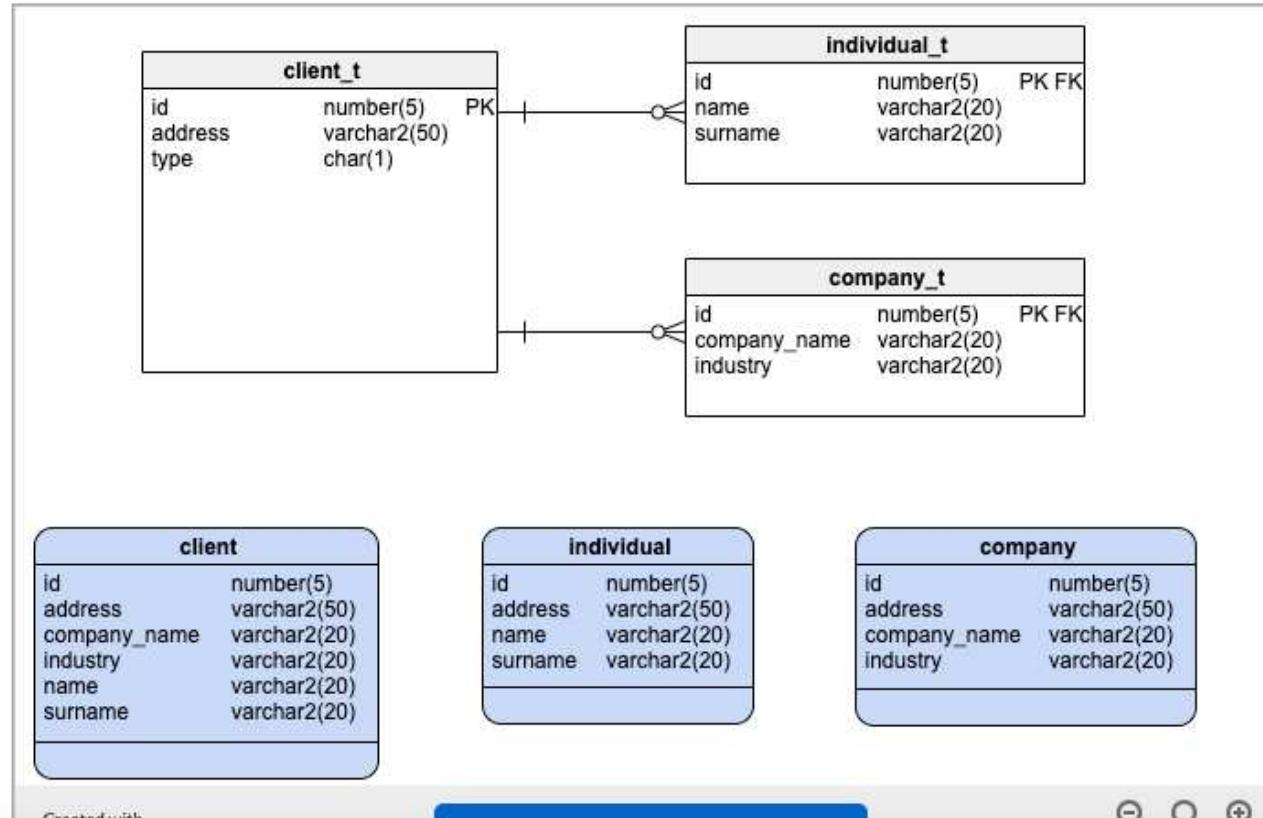
Two-table implementation

In a two-table implementation, we create a table for each of the subtypes. Each table gets a column for all attributes of the supertype and also a column for each attribute belonging to the subtype. Access to information in this situation is limited, that's why it is important to create a view that is the union of the tables. We can add an additional attribute called 'type' that describes the subtype.

The diagram below presents two tables, `individual` and `company`, and a view (the blue one) called `client`.



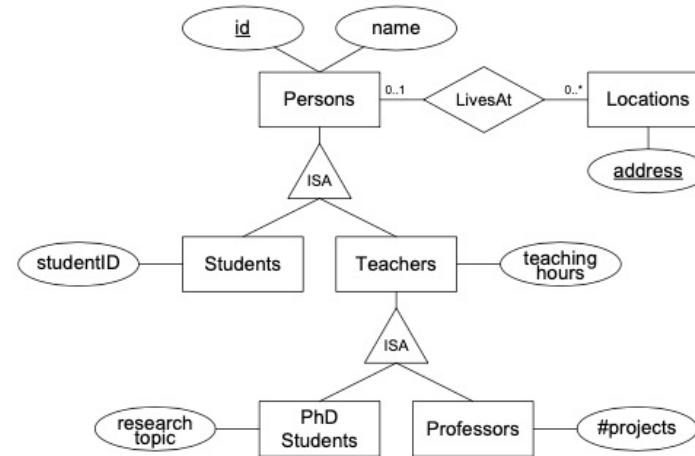
Three Table



ISA Relationship



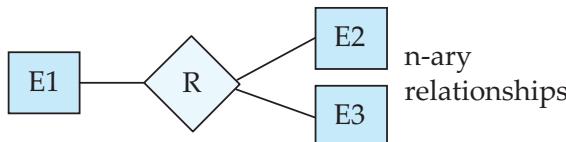
ISA Relationship





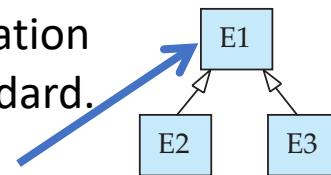
ER vs. UML Class Diagrams

ER Diagram Notation

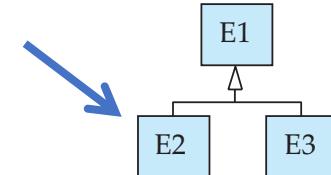


n-ary
relationships

I use this approach
in Crow's Foot Notation
but that is not standard.

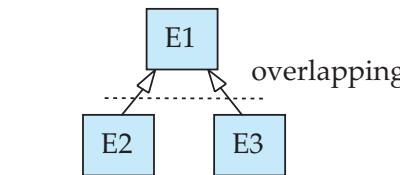
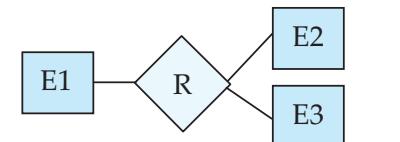


overlapping
generalization

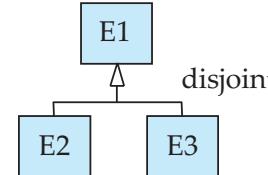


disjoint
generalization

Equivalent in UML



overlapping



disjoint

- * Generalization can use merged or separate arrows independent of disjoint/overlapping

Relation Model and Algebra

Relation Model and Algebra

Just kidding!
We are basically done with relational algebra!

SQL

Metadata

Codd's 12 Rules

Rule 1: Information Rule

The data stored in a database, may it be user data or metadata, must be a value of some table cell. Everything in a database must be stored in a table format.

Rule 2: Guaranteed Access Rule

Every single data element (value) is guaranteed to be accessible logically with a combination of table-name, primary-key (row value), and attribute-name (column value). No other means, such as pointers, can be used to access data.

Rule 3: Systematic Treatment of NULL Values

The NULL values in a database must be given a systematic and uniform treatment. This is a very important rule because a NULL can be interpreted as one the following – data is missing, data is not known, or data is not applicable.

Rule 4: Active Online Catalog

The structure description of the entire database must be stored in an online catalog, known as data dictionary, which can be accessed by authorized users. Users can use the same query language to access the catalog which they use to access the database itself.

Rule 5: Comprehensive Data Sub-Language Rule

A database can only be accessed using a language having linear syntax that supports data definition, data manipulation, and transaction management operations. This language can be used directly or by means of some application. If the database allows access to data without any help of this language, then it is considered as a violation.

Rule 6: View Updating Rule

All the views of a database, which can theoretically be updated, must also be updatable by the system.

Codd's 12 Rules

Rule 7: High-Level Insert, Update, and Delete Rule

A database must support high-level insertion, updation, and deletion. This must not be limited to a single row, that is, it must also support union, intersection and minus operations to yield sets of data records.

Rule 8: Physical Data Independence

The data stored in a database must be independent of the applications that access the database. Any change in the physical structure of a database must not have any impact on how the data is being accessed by external applications.

Rule 9: Logical Data Independence

The logical data in a database must be independent of its user's view (application). Any change in logical data must not affect the applications using it. For example, if two tables are merged or one is split into two different tables, there should be no impact or change on the user application. This is one of the most difficult rules to apply.

Rule 10: Integrity Independence

A database must be independent of the application that uses it. All its integrity constraints can be independently modified without the need of any change in the application. This rule makes a database independent of the front-end application and its interface.

Rule 11: Distribution Independence

The end-user must not be able to see that the data is distributed over various locations. Users should always get the impression that the data is located at one site only. This rule has been regarded as the foundation of distributed database systems.

Rule 12: Non-Subversion Rule

If a system has an interface that provides access to low-level records, then the interface must not be able to subvert the system and bypass security and integrity constraints.

Metadata and Catalog

- ‘Metadata is "data that provides information about other data". In other words, it is "data about data". Many distinct types of metadata exist, including descriptive metadata, structural metadata, administrative metadata, reference metadata and statistical metadata.’
(<https://en.wikipedia.org/wiki/Metadata>)
- “The database catalog of a database instance consists of metadata in which definitions of database objects such as base tables, views (virtual tables), synonyms, value ranges, indexes, users, and user groups are stored.”

The SQL standard specifies a uniform means to access the catalog, called the INFORMATION_SCHEMA, but not all databases follow this ...”

(https://en.wikipedia.org/wiki/Database_catalog)

- Codd’s Rule 4: Dynamic online catalog based on the relational model:
 - The data base description is represented at the logical level in the same way as ordinary data, so that authorized users can apply the same relational language to its interrogation as they apply to the regular data.

Data Definition Language (DDL)

- Specification notation for defining the database schema

Example:

```
create table instructor (
    ID      char(5),
    name    varchar(20),
    dept_name varchar(20),
    salary   numeric(8,2))
```

- DDL compiler generates a set of table templates stored in a ***data dictionary***
- Data dictionary contains metadata (i.e., data about data)
 - Database schema
 - Integrity constraints
 - Primary key (ID uniquely identifies instructors)
 - Authorization
 - Who can access what

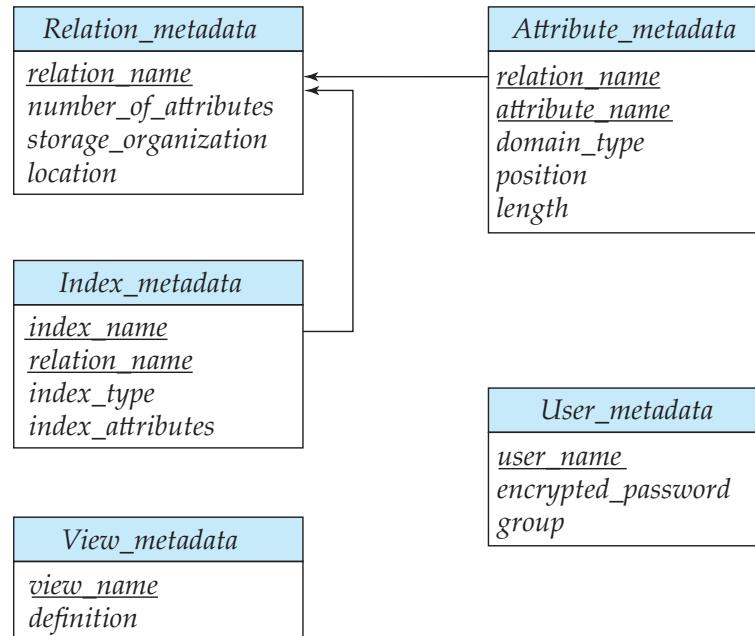
Data Dictionary Storage

The **Data dictionary** (also called **system catalog**) stores **metadata**; that is, data about data, such as

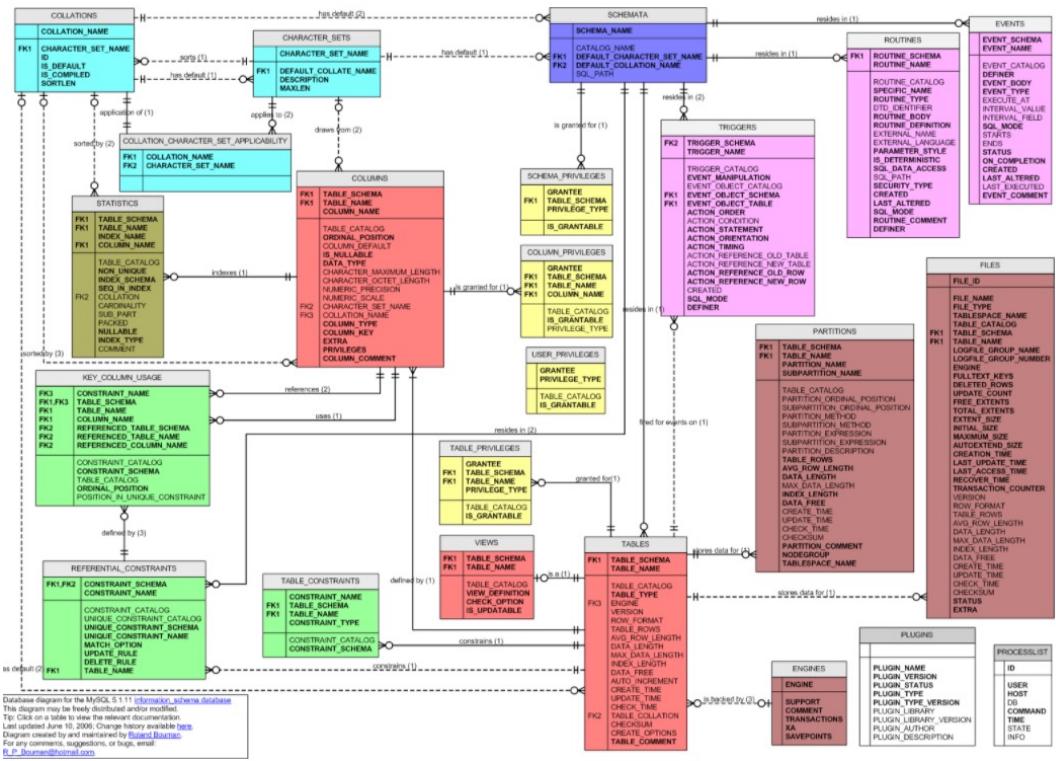
- Information about relations
 - names of relations
 - names, types and lengths of attributes of each relation
 - names and definitions of views
 - integrity constraints
- User and accounting information, including passwords
- Statistical and descriptive data
 - number of tuples in each relation
- Physical file organization information
 - How relation is stored (sequential/hash/...)
 - Physical location of relation
- Information about indices (Chapter 14)

Relational Representation of System Metadata

- Relational representation on disk
- Specialized data structures designed for efficient access, in memory



MySQL Catalog (Information_Schema)



Some of the MySQL Information Schema Tables:

- 'ADMINISTRABLE_ROLE_AUTHORIZATIONS'
 - 'APPLICABLE_ROLES'
 - 'CHARACTER_SETS'
 - 'CHECK_CONSTRAINTS'
 - 'COLUMN_PRIVILEGES'
 - 'COLUMN_STATISTICS'
 - 'COLUMNS'
 - 'ENABLED_ROLES'
 - 'ENGINES'
 - 'EVENTS'
 - 'FILES'
 - 'KEY_COLUMN_USAGE'
 - 'PARAMETERS'
 - 'REFERENTIAL_CONSTRAINTS'
 - 'RESOURCE_GROUPS'
 - 'ROLE_COLUMN_GRANTS'
 - 'ROLE_ROUTINE_GRANTS'
 - 'ROLE_TABLE_GRANTS'
 - 'ROUTINES'
 - 'SCHEMA_PRIVILEGES'
 - 'STATISTICS'
 - 'TABLE_CONSTRAINTS'
 - 'TABLE_PRIVILEGES'
 - 'TABLES'
 - 'TABLESPACES'
 - 'TRIGGERS'
 - 'USER_PRIVILEGES'
 - 'VIEW_ROUTINE_USAGE'
 - 'VIEW_TABLE_USAGE'
 - 'VIEWS'
 - CREATE and ALTER statements modify the data.
 - DBMS reads information:
 - Parsing
 - Optimizer
 - etc.

Metadata

Notebook examples.

Some Types and Functions

Functions

MySQL CHEAT SHEET: STRING FUNCTIONS

by sqlbackupandftp.com with ♥

MEASUREMENT

Return a string containing binary representation of a number

`BIN (12) = '1100'`

Return length of argument in bits

`BIT_LENGTH ('MySQL') = 40`

Return number of characters in argument

`CHAR_LENGTH ('MySQL') = 5`

`CHARACTER_LENGTH ('MySQL') = 5`

Return the length of a string in bytes

`LENGTH ('Ø') = 2`

`LENGTH ('A') = 1`

`OCTET_LENGTH ('Ø') = 2`

`OCTET_LENGTH ('X') = 1`

Return a soundex string

`SOUNDEX ('MySQL') = 'M240'`

`SOUNDEX ('MySQLDatabase') = 'M24312'`

Compare two strings

`STRCMP ('A', 'A') = 0`

`STRCMP ('A', 'B') = -1`

`STRCMP ('B', 'A') = 1`

SEARCH

Return the index of the first occurrence of substring

`INSTR ('MySQL', 'Sql') = 3`

`INSTR ('Sql', 'MySQL') = 0`

Return the position of the first occurrence of substring

`LOCATE ('Sql', 'MySQLSql') = 3`

`LOCATE ('xSql', 'MySQL') = 0`

`LOCATE ('Sql', 'MySQLSql', 5) = 6`

`POSITION('Sql' IN 'MySQLSql') = 3`

Pattern matching using regular expressions

`'abc' RLIKE '[a-z]+ = 1`

`'123' RLIKE '[a-z]+ = 0`

Return a substring from a string before the specified number of occurrences of the delimiter

`SUBSTRING_INDEX ('A:B:C', ':', 1) = 'A'`

`SUBSTRING_INDEX ('A:B:C', ':', 2) = 'A:B'`

`SUBSTRING_INDEX ('A:B:C', ':', -2) = 'B:C'`

CONVERSION

Return numeric value of left-most character

`ASCII ('2') = 50`

`ASCII (2) = 50`

`ASCII ('ð') = 100`

Return the character for each number passed

`CHAR (45*256+45) = CHAR (45,45) = '-'`

`CHARSET(CHAR (X'65' USING utf8)) = 'utf8'`

Decode to / from a base-64 string

`TO_BASE64 ('abc') = 'YnVz'`

`FROM_BASE64 ('YnVz') = 'abc'`

Convert string or number to its hexadecimal representation

`X'616263' = 'abc'`

`HEX ('abc') = 616263`

`HEX(255) = 'FF'`

`CONV(HEX(255), 16, 10) = 255`

Convert each pair of hexadecimal digits to a character

`UNHEX ('4D7953514C') = 'MySQL'`

`UNHEX ('GG') = NULL`

`UNHEX (HEX ('abc')) = 'abc'`

Return the argument in lowercase

`LOWER ('MySQL') = 'mysql'`

`LCASE ('MySQL') = 'mysql'`

Load the named file

`SET blob_col=LOAD_FILE ('/tmp/picture')`

Return a string containing octal representation of a number

`OCT (12) = '14'`

Return character code for leftmost character of the argument

`ORD ('2') = 50`

Escape the argument for use in an SQL statement

`QUOTE ('Don\'t!') = 'Don\'t!'`

`QUOTE (NULL) = 'NULL'`

Convert to uppercase

`UPPER ('mysql') = 'MYSQL'`

`UCASE ('mysql') = 'MYSQL'`

MODIFICATION

Return concatenated string

`CONCAT ('My', ' ', 'SQL') = 'MySQL'`

`CONCAT ('My', NULL, 'SQL') = NULL`

`CONCAT ('14.3') = '14.3'`

Return concatenate with separator

`CONCAT_WS (' ', 'My', 'Sql') = 'My,Sql'`

`CONCAT_WS (' ', 'My', NULL, 'Sql') = 'My,Sql'`

Return a number formatted to specified number of decimal places

`FORMAT ('12332.123456, 4) = 12,332.1235`

`FORMAT ('1232.1, 4) = 12,32,1000`

`FORMAT ('1232.2, 0) = 1232,2`

`FORMAT ('1232.2, 2, 'de_DE') = 12.332,20`

Insert a substring at the specified position up to the specified number of characters

`INSERT ('12345', 3, 2, 'ABC') = '12ABC5'`

`INSERT ('12345', 10, 2, 'ABC') = '12345'`

`INSERT ('12345', 3, 10, 'ABC') = '12ABC'`

Return the leftmost number of characters as specified

`LEFT ('MySQL', 2) = 'My'`

Return the string argument, left-padded with the specified string

`LPAD ('Sql', 2, ':') = 'S:q'`

`LPAD ('Sql', 4, ':') = 'S:ql'`

`LPAD ('Sql', 5, ':') = ':):)Sql'`

Remove leading spaces

`LTRIM (' ' MySQL) = 'MySQL'`

Repeat a string the specified number of times

`REPEAT ('MySQL', 3) = 'MySQLMySQLMySQL'`

Replace occurrences of a specified string

`REPLACE ('NoSql', 'No', 'My') = 'MySql'`

Reverse the characters in a string

`REVERSE ('MySQL') = 'lqSylM'`

Return the specified rightmost number of characters

`RIGHT ('MySQL', 3) = 'Sql'`

Return the string argument, right-padded with the specified string

`RPAD ('Sql', 2, ':') = 'Sql'`

`RPAD ('Sql', 4, ':') = 'Sql:'`

`RPAD ('Sql', 7, ':') = 'Sql:;;:;'`

Remove trailing spaces

`RTRIM ('MySQL ') = 'MySQL'`

Return a string of the specified number of spaces

`SPACE ('6') = ''`

Return the substring as specified

`SUBSTRING=SUBSTR-MID ('MySQL',3) = 'Sql'`

`SUBSTRING=SUBSTR-MID ('MySQL', FROM 4) = 'ql'`

`SUBSTRING=SUBSTR-MID ('MySQL',3,1) = 'S'`

`SUBSTRING=SUBSTR-MID ('MySQL',-3) = 'Sql'`

`SUBSTRING=SUBSTR-MID ('MySQL', FROM -4 FOR 2) = 'yS'`

Remove leading and trailing spaces

`TRIM(' MySql ') = ' MySql '`

`TRIM(LEADING 'x' FROM 'xxxSqlMy') = 'MySql'`

`TRIM(BOTH 'My' FROM 'MySqlMy') = 'Sql'`

`TRIM(TRAILING 'Sql' FROM 'MySql') = 'My'`

SETS

Return string at index number

`ELT (1, 'ej', 'Heja', 'hej', 'foo') = 'ej'`

`ELT (4, 'ej', 'Heja', 'hej', 'foo') = 'foo'`

Return a string such that for every bit set in the value bits, you get an on string and for every unset bit, you get an off string

`EXPORT_SET (5, 'Y','N','N','N',4) = 'Y,N,Y,N'`

`EXPORT_SET (6, '1','0','0','0',6) = '0,1,1,0,0,0'`

Return the index (position) of the first argument in the subsequent arguments

`FIELD ('ej','Hj','ej','Heja','hej','oo') = 2`

`FIELD ('fo','Hj','ej','Heja','hej','oo') = 0`

Return the index position of the first argument within the second argument

`FIND_IN_SET ('b', 'a,b,c,d') = 2`

`FIND_IN_SET ('z', 'a,b,c,d') = 0`

`FIND_IN_SET ('a', 'a,b,c,d') = 0`

Return a set of comma-separated strings that have the corresponding bit in bits set

`MAKE_SET (1, 'a','b','c') = 'a'`

`MAKE_SET (1|4,'ab','cd','ef') = 'ab,ef'`

`MAKE_SET (1|4,'ab','cd',NULL,'ef') = 'ab'`

`MAKE_SET (0, 'a','b','c') = ''`

TEXT FUNCTIONS

CONCATENATION

Use the || operator to concatenate two strings:
`SELECT 'Hi ' || 'there!';`
-- result: Hi there!

Remember that you can concatenate only character strings using |||. Use this trick for numbers:
`SELECT '' || 4 || 2;`
-- result: 42

Some databases implement non-standard solutions for concatenating strings like CONCAT() or CONCAT_WS(). Check the documentation for your specific database.

LIKE OPERATOR – PATTERN MATCHING

Use the _ character to replace any single character. Use the % character to replace any number of characters (including 0 characters).

Fetch all names that start with any letter followed by 'atherine':

```
SELECT name
FROM names
WHERE name LIKE '_atherine';
Fetch all names that end with 'a':
SELECT name
FROM names
WHERE name LIKE '%a';

```

USEFUL FUNCTIONS

Get the count of characters in a string:

```
SELECT LENGTH('LearnSQL.com');
-- result: 12
```

Convert all letters to lowercase:

```
SELECT LOWER('LEARNSQL.COM');
-- result: learnsql.com
```

Convert all letters to uppercase:

```
SELECT UPPER('LearnSQL.com');
-- result: LEARNSQL.COM
```

Convert all letters to lowercase and all first letters to uppercase (not implemented in MySQL and SQL Server):

```
SELECT INITCAP('edgar frank ted codd');
-- result: Edgar Frank Ted Codd
```

Get just a part of a string:

```
SELECT SUBSTRING('LearnSQL.com', 9);
-- result: .com
SELECT SUBSTRING('LearnSQL.com', 0, 6);
-- result: Learn
Replace part of a string:
SELECT REPLACE('LearnSQL.com', 'SQL',
'Python');
-- result: LearnPython.com
```

NUMERIC FUNCTIONS

BASIC OPERATIONS

Use +, -, *, / to do some basic math. To get the number of seconds in a week:

```
SELECT 60 * 60 * 24 * 7; -- result: 604800
```

CASTING

From time to time, you need to change the type of a number. The CAST() function is there to help you out. It lets you change the type of almost anything (integer, numeric, double precision, varchar, and many more).

Get the number as an integer (without rounding):

```
SELECT CAST(1234.567 AS integer);
-- result: 1234
```

Change a column type to double precision

```
SELECT CAST(column AS double precision);
```

USEFUL FUNCTIONS

Get the remainder of a division:

```
SELECT MOD(13, 2);
-- result: 1
```

Round a number to its nearest integer:

```
SELECT ROUND(1234.56789);
-- result: 1235
```

Round a number to three decimal places:

```
SELECT ROUND(1234.56789, 3);
-- result: 1234.568
```

PostgreSQL requires the first argument to be of the type numeric – cast the number when needed.

To round the number up:

```
SELECT CEIL(13.8); -- result: 14
SELECT CEIL(-13.9); -- result: -13
```

The CEIL(x) function returns the **smallest** integer **not less** than x. In SQL Server, the function is called CEILING().

To round the number down:

```
SELECT FLOOR(13.8); -- result: 13
SELECT FLOOR(-13.2); -- result: -14
```

The FLOOR(x) function returns the **greatest** integer **not greater** than x.

To round towards 0 irrespective of the sign of a number:

```
SELECT TRUNC(13.5); -- result: 13
SELECT TRUNC(-13.5); -- result: -13
```

TRUNC(x) works the same way as CAST(x AS integer). In MySQL, the function is called TRUNCATE().

To get the absolute value of a number:

```
SELECT ABS(-12); -- result: 12
```

To get the square root of a number:

```
SELECT SQRT(9); -- result: 3
```

NULLS

To retrieve all rows with a missing value in the price column:

```
WHERE price IS NULL
```

To retrieve all rows with the weight column populated:

```
WHERE weight IS NOT NULL
```

Why shouldn't you use price = NULL or weight != NULL? Because databases don't know if those expressions are true or false – they are evaluated as NULLs.

Moreover, if you use a function or concatenation on a column that is NULL in some rows, then it will get propagated. Take a look:

domain	LENGTH(domain)
LearnSQL.com	12
LearnPython.com	15
NULL	NULL
vertabelo.com	13

USEFUL FUNCTIONS

COALESCE(x, y, ...)

To replace NULL in a query with something meaningful:

```
SELECT
  domain,
  COALESCE(domain, 'domain missing')
FROM contacts;
```

domain	coalesce
LearnSQL.com	LearnSQL.com
NULL	domain missing

The COALESCE() function takes any number of arguments and returns the value of the first argument that isn't NULL.

NULIF(x, y)

To save yourself from division by 0 errors:

```
SELECT
  last_month,
  this_month,
  this_month * 100.0
  / NULIF(last_month, 0)
  AS better_by_percent
FROM video_views;
```

last_month	this_month	better_by_percent
72376	1085679	150.0
0	178123	NULL

The NULIF(x, y) function will return NULL if x is the same as y, else it will return the x value.

CASE WHEN

The basic version of CASE WHEN checks if the values are equal (e.g., if fee is equal to 50, then 'normal' is returned). If there isn't a matching value in the CASE WHEN, then the ELSE value will be returned (e.g., if fee is equal to 49, then 'not available' will show up).

```
SELECT
  CASE fee
    WHEN 50 THEN 'normal'
    WHEN 10 THEN 'reduced'
    WHEN 0 THEN 'free'
    ELSE 'not available'
  END AS tariff
FROM ticket_types;
```

The most popular type is the **searched CASE WHEN** – it lets you pass conditions (as you'd write them in the WHERE clause), evaluates them in order, then returns the value for the first condition met.

```
SELECT
  CASE
    WHEN score >= 90 THEN 'A'
    WHEN score > 60 THEN 'B'
    ELSE 'F'
  END AS grade
FROM test_results;
```

Here, all students who scored at least 90 will get an A, those with the score above 60 (and below 90) will get a B, and the rest will receive an F.

TROUBLESHOOTING

Integer division

When you don't see the decimal places you expect, it means that you are dividing between two integers. Cast one to decimal:

```
CAST(123 AS decimal) / 2
```

Division by 0
To avoid this error, make sure that the denominator is not equal to 0. You can use the NULIF() function to replace 0 with a NULL, which will result in a NULL for the whole expression:

```
count / NULIF(count_all, 0)
```

Inexact calculations

If you do calculations using real (floating point) numbers, you'll end up with some inaccuracies. This is because this type is meant for scientific calculations such as calculating the velocity. Whenever you need accuracy (such as dealing with monetary values), use the decimal/numeric type (or money if available).

Errors when rounding with a specified precision

Most databases won't complain, but do check the documentation if they do. For example, if you want to specify the rounding precision in PostgreSQL, the value must be of the numeric type.

Try out the interactive **Standard SQL Functions** course at LearnSQL.com, and check out our other SQL courses.

LearnSQL.com is owned by Vertabelo SA
vertabelo.com | CC-BY-NC-ND Vertabelo SA

Functions

Standard SQL Functions Cheat Sheet

AGGREGATION AND GROUPING

- `COUNT(expr)` – the count of values for the rows within the group
- `SUM(expr)` – the sum of values within the group
- `AVG(expr)` – the average value for the rows within the group
- `MIN(expr)` – the minimum value within the group
- `MAX(expr)` – the maximum value within the group

To get the number of rows in the table:

```
SELECT COUNT(*)  
FROM city;
```

To get the number of non-NULL values in a column:

```
SELECT COUNT(rating)  
FROM city;
```

To get the count of unique values in a column:

```
SELECT COUNT(DISTINCT country_id)  
FROM city;
```

GROUP BY

CITY	
name	country_id
Paris	1
Marseille	1
Lyon	1
Berlin	2
Hamburg	2
Munich	2
Warsaw	4
Cracow	4

→

CITY	
country_id	count
1	3
2	3
4	2

The example above – the count of cities in each country:

```
SELECT name, COUNT(country_id)  
FROM city  
GROUP BY name;
```

The average rating for the city:

```
SELECT city_id, AVG(rating)  
FROM ratings  
GROUP BY city_id;
```

Common mistake: COUNT(*) and LEFT JOIN

When you join the tables like this: `client LEFT JOIN project`, and you want to get the number of projects for every client you know, `COUNT(*)` will return 1 for each client even if you've never worked for them. This is because, they're still present in the list but with the NULL in the fields related to the project after the JOIN. To get the correct count (0 for the clients you've never worked for), count the values in a column of the other table, e.g., `COUNT(project_name)`. Check out this [exercise](#) to see an example.

DATE AND TIME

There are 3 main time-related types: `date`, `time`, and `timestamp`. Time is expressed using a 24-hour clock, and it can be as vague as just hour and minutes (e.g., 15:30 – 3:30 p.m.) or as precise as microseconds and time zone (as shown below):

2021-12-31 14:39:53.662522-05



YYYY-mm-dd HH:MM:SS.#####TZ

14:39:53.662522-05 is almost 2:40 p.m. CDT (e.g., in Chicago; in UTC it'd be 7:40 p.m.). The letters in the above example represent:

In the date part:

- YYYY – the 4-digit year.
 - mm – the zero-padded month (01–January through 12–December).
 - dd – the zero-padded day.
- HH – the zero-padded hour in a 24-hour clock.
 - MM – the minutes.
 - SS – the seconds. *Optional*.
 - ##### – the smaller parts of a second – they can be expressed using 1 to 6 digits. *Optional*.
 - ±TZ – the timezone. It must start with either + or -, and use two digits relative to UTC. *Optional*.

What time is it?

To answer that question in SQL, you can use:

- `CURRENT_TIME` – to find what time it is.
- `CURRENT_DATE` – to get today's date. (`GETDATE()` in SQL Server.)
- `CURRENT_TIMESTAMP` – to get the timestamp with the two above.

Creating values

To create a date, time, or timestamp, simply write the value as a string and cast it to the proper type.

```
SELECT CAST('2021-12-31' AS date);  
SELECT CAST('15:31' AS time);  
SELECT CAST('2021-12-31 23:59:29+02' AS timestamp);
```

```
SELECT CAST('15:31.124769' AS time);
```

Be careful with the last example – it will be interpreted as 15 minutes 31 seconds and 124769 microseconds! It is always a good idea to write 00 explicitly for hours: '00:15:31.124769'.

You might skip casting in simple conditions – the database will know what you mean.

```
SELECT airline, flight_number, departure_time  
FROM airport_schedule  
WHERE departure_time < '12:00';
```

INTERVALS

Note: In SQL Server, intervals aren't implemented – use the `DATEADD()` and `DATEDIFF()` functions.

To get the simplest interval, subtract one time value from another:

```
SELECT CAST('2021-12-31 23:59:59' AS timestamp) - CAST('2021-06-01 12:00:00' AS timestamp);  
-- result: 213 days 11:59:59
```

To define an interval: `INTERVAL '1' DAY`

This syntax consists of three elements: the `INTERVAL` keyword, a quoted value, and a time part keyword (singular form). You can use the following time parts: YEAR, MONTH, WEEK, DAY, HOUR, MINUTE, and SECOND. In MySQL, omit the quotes. You can join many different INTERVALs using the + or – operator:

```
INTERVAL '1' YEAR + INTERVAL '3' MONTH
```

In some databases, there's an easier way to get the above value. And it accepts plural forms! `INTERVAL '1 year 3 months'`

There are two more syntaxes in the Standard SQL:

Syntax	What it does
INTERVAL 'x-y' YEAR TO MONTH	INTERVAL 'x' year y month'
INTERVAL 'x-y' DAY TO SECOND	INTERVAL 'x' day y second'

In MySQL, write `year_month` instead of `YEAR TO MONTH` and `day_second` instead of `DAY TO SECOND`.

To get the last day of a month, add one month and subtract one day:

```
SELECT CAST('2021-02-01' AS date) + INTERVAL '1' MONTH - INTERVAL '1' DAY;
```

To get all events for next three months from today:

```
SELECT event_date, event_name  
FROM calendar  
WHERE event_date BETWEEN CURRENT_DATE AND  
CURRENT_DATE + INTERVAL '3' MONTH;
```

To get part of the date:

```
SELECT EXTRACT(YEAR FROM birthday)  
FROM artists;
```

One of possible returned values: 1946. In SQL Server, use the `DATEPART(part, date)` function.

TIME ZONES

In the SQL Standard, the `date` type can't have an associated time zone, but the `time` and `timestamp` types can. In the real world, time zones have little meaning without the date, as the offset can vary through the year because of `daylight saving time`. So, it's best to work with the `timestamp` values.

When working with the type `timestamp` with time zone (abbr. `timestamptz`), you can type in the value in your local time zone, and it'll get converted to the UTC time zone as it is inserted into the table. Later when you select from the table it gets converted back to your local time zone. This is immune to time zone changes.

AT TIME ZONE

To operate between different time zones, use the `AT TIME ZONE` keyword.

If you use this format: {`timestamp` without time zone} `AT TIME ZONE` {time zone}, then the database will read the time stamp in the specified time zone and convert it to the time zone local to the display. It returns the time in the format `timestamp with time zone`.

If you use this format: {`timestamp` with time zone} `AT TIME ZONE` {time zone}, then the database will convert the time in one time zone to the target time zone specified by `AT TIME ZONE`. It returns the time in the format `timestamptz` without time zone, in the target time zone.

You can define the time zone with popular shortcuts like UTC, MST, or GMT, or by continent/city such as: America/New_York, Europe/London, and Asia/Tokyo.

Examples

Set the local time zone to 'America/New_York'.

```
SELECT TIMESTAMP '2021-07-16 21:00:00' AT  
TIME ZONE 'America/Los_Angeles';  
-- result: 2021-07-17 00:00:00-04
```

Here, the database takes a timestamp without a time zone and it's told it's in Los Angeles time, which is then converted to the local time – New York for displaying. This answers the question "`At what time should I turn on the TV if the show starts at 9 PM in Los Angeles?`"

```
SELECT TIMESTAMP WITH TIME ZONE '2021-06-20  
19:30:00' AT TIME ZONE 'Australia/Sydney';  
-- result: 2021-06-21 09:30:00
```

Here, the database gets a timestamp specified in the local time zone and converts it to the time in Sydney (note that it didn't return a time zone). This answers the question "`What time is it in Sydney if it's 7:30 PM here?`"

Functions

- There are dozens if not hundreds of standard functions in SQL.
- All DBMS implementations have product specific functions.
- General rule:
 - If you have to do something, ask yourself
 - Am I the first one who ever had to do this?
 - If the answer is “No,” then ask Dr. Google/ChatGPT.
 - If the answer is Yes,” ask yourself, “Am I sure this is a good idea.”
- The functions are useful and straightforward.
- Some examples → To the notebook we go, yo ho!

Views



Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know an instructors name and department, but not the salary. This person should see a relation described, in SQL, by

```
select ID, name, dept_name  
from instructor
```

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a “virtual relation” is called a **view**.



View Definition

- A view is defined using the **create view** statement which has the form

create view *v* as <query expression>

where <query expression> is any legal SQL expression. The view name is represented by *v*.

- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- View definition is not the same as creating a new relation by evaluating the query expression
 - Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view.



View Definition and Use

- A view of instructors without their salary

```
create view faculty as  
    select ID, name, dept_name  
        from instructor
```

- Find all instructors in the Biology department

```
select name  
    from faculty  
    where dept_name = 'Biology'
```

- Create a view of department salary totals

```
create view departments_total_salary(dept_name, total_salary) as  
    select dept_name, sum (salary)  
        from instructor  
    group by dept_name;
```



Views Defined Using Other Views

- One view may be used in the expression defining another view
- A view relation v_1 is said to **depend directly** on a view relation v_2 if v_2 is used in the expression defining v_1
- A view relation v_1 is said to **depend on** view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2
- A view relation v is said to be **recursive** if it depends on itself.



Views Defined Using Other Views

- **create view *physics_fall_2017* as**
select course.course_id, sec_id, building, room_number
from course, section
where course.course_id = section.course_id
and course.dept_name = 'Physics'
and section.semester = 'Fall'
and section.year = '2017';
- **create view *physics_fall_2017_watson* as**
select course_id, room_number
from *physics_fall_2017*
where building= 'Watson';



View Expansion

- Expand the view :

```
create view physics_fall_2017_watson as
    select course_id, room_number
        from physics_fall_2017
    where building= 'Watson'
```

- To:

```
create view physics_fall_2017_watson as
    select course_id, room_number
        from (select course.course_id, building, room_number
              from course, section
             where course.course_id = section.course_id
               and course.dept_name = 'Physics'
               and section.semester = 'Fall'
               and section.year = '2017')
    where building= 'Watson';
```



View Expansion (Cont.)

- A way to define the meaning of views defined in terms of other views.
- Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations.
- View expansion of an expression repeats the following replacement step:

repeat
 Find any view relation v_i in e_1
 Replace the view relation v_i by the expression defining v_i
until no more view relations are present in e_1
- As long as the view definitions are not recursive, this loop will terminate



Materialized Views

- Certain database systems allow view relations to be physically stored.
 - Physical copy created when the view is defined.
 - Such views are called **Materialized view**:
- If relations used in the query are updated, the materialized view result becomes out of date
 - Need to **Maintain** the view, by updating the view whenever the underlying relations are updated.



Update of a View

- Add a new tuple to *faculty* view which we defined earlier

```
insert into faculty
```

```
values ('30765', 'Green', 'Music');
```

- This insertion must be represented by the insertion into the *instructor* relation

- Must have a value for salary.

- Two approaches

- Reject the insert
 - Insert the tuple

```
('30765', 'Green', 'Music', null)
```

into the *instructor* relation



Some Updates Cannot be Translated Uniquely

- **create view** *instructor_info* **as**
select *ID, name, building*
from *instructor, department*
where *instructor.dept_name= department.dept_name;*
- **insert into** *instructor_info*
values ('69987', 'White', 'Taylor');
- Issues
 - Which department, if multiple departments in Taylor?
 - What if no department is in Taylor?



And Some Not at All

- ```
create view history_instructors as
 select *
 from instructor
 where dept_name= 'History';
```
- What happens if we insert  
('25566', 'Brown', 'Biology', 100000)  
into *history\_instructors*?



# View Updates in SQL

- Most SQL implementations allow updates only on simple views
  - The **from** clause has only one database relation.
  - The **select** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or **distinct** specification.
  - Any attribute not listed in the **select** clause can be set to null
  - The query does not have a **group by** or **having** clause.

# *Integrity Constraints*



# Integrity Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
  - A checking account must have a balance greater than \$10,000.00
  - A salary of a bank employee must be at least \$4.00 an hour
  - A customer must have a (non-null) phone number

## DFF

- Without integrity constraints in the database, maintaining data correctness requires:
  - Lots of users know what to do and do not make mistakes.
  - Dozens of programs correctly implement constraints in the code and stay up to date on changes.
- Implementing the constraints as part of the schema eliminates many issues.



# Constraints on a Single Relation

- **not null**
- **primary key**
- **unique**
- **check (P)**, where P is a predicate



# Not Null Constraints

- **not null**
  - Declare *name* and *budget* to be **not null**  
*name varchar(20) not null*  
*budget numeric(12,2) not null*



# Unique Constraints

- **unique** ( $A_1, A_2, \dots, A_m$ )
  - The unique specification states that the attributes  $A_1, A_2, \dots, A_m$  form a candidate key.
  - Candidate keys are permitted to be null (in contrast to primary keys).

# Simple Example

- Consider a simple example of an entity class *major*:
  - *major(id, name, track)*
  - “id” is a uniquely generated ID
  - “name” is the major name, e.g. “Computer Science,” “Economics,” ... ...
  - “track” is a sub-track/specialty within the major, e.g. “Applications,” “AI/ML,” ...
    - “track” is optional
    - The combination of *(name, track)* is *unique*.
- **Note:** In many DBMS, this automatically creates indices for keys/constraints.
- Switch to Notebook.

```
create table if not exists majors
(
 id int auto_increment
 primary key,
 major_name varchar(64) not null,
 major_track varchar(64) null,
 constraint table_name_pk
 unique (major_name, major_track)
);
```



# The check clause

- The **check** (P) clause specifies a predicate P that must be satisfied by every tuple in a relation.
- Example: ensure that semester is one of fall, winter, spring or summer

```
create table section
 (course_id varchar (8),
 sec_id varchar (8),
 semester varchar (6),
 year numeric (4,0),
 building varchar (15),
 room_number varchar (7),
 time_slot_id varchar (4),
 primary key (course_id, sec_id, semester, year),
 check (semester in ('Fall', 'Winter', 'Spring', 'Summer')))
```

DFF:

- We could handle the *semester check* with an *enum*.
- Switch to notebook for a slightly different example.



# Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
  - Example: If “Biology” is a department name appearing in one of the tuples in the *instructor* relation, then there exists a tuple in the *department* relation for “Biology”.
- Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the primary key of S. A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S.



## Referential Integrity (Cont.)

- Foreign keys can be specified as part of the SQL **create table** statement  
**foreign key (*dept\_name*) references *department***
- By default, a foreign key references the primary-key attributes of the referenced table.
- SQL allows a list of attributes of the referenced relation to be specified explicitly.  
**foreign key (*dept\_name*) references *department* (*dept\_name*)**



# Cascading Actions in Referential Integrity

- When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.
- An alternative, in case of delete or update is to cascade

```
create table course (
 ...
 dept_name varchar(20),
 foreign key (dept_name) references department
 on delete cascade
 on update cascade,
 ...
)
```

- Instead of cascade we can use :
  - **set null**,
  - **set default**

## DFF:

- I do not like using *cascade*. I think making changes should be explicit.
- Other people disagree.
- You will get some simple practice on HW or exams.

# *Indexes*

## *Review, Concepts, Examples*



# Index Creation

- Many queries reference only a small proportion of the records in a table.
- It is inefficient for the system to read every record to find a record with particular value
- An **index** on an attribute of a relation is a data structure that allows the database system to find those tuples in the relation that have a specified value for that attribute efficiently, without scanning through all the tuples of the relation.
- We create an index with the **create index** command

```
create index <name> on <relation-name> (attribute);
```



# Index Creation Example

- **create table student**  
*(ID varchar (5),  
name varchar (20) not null,  
dept\_name varchar (20),  
tot\_cred numeric (3,0) default 0,  
primary key (ID))*
- **create index studentID\_index on student(ID)**
- The query:

```
select *
from student
where ID = '12345'
```

can be executed by using the index to find the required record, without looking at all records of *student*

DFF:

- An index on (column1, column2, column3)
- Is also an index on (column1) and (column1, column2)
- But not (column2), (column3), (column2, column3).
- We will see “why” later in the semester.

# *REST*

# Data Modeling Concepts and REST

Almost any data model has the same core concepts:

- Types and instances:
  - Entity Type: A definition of a type of thing with properties and relationships.
  - Entity Instance: A specific instantiation of the Entity Type
  - Entity Set Instance: An Entity Type that:
    - Has properties and relationships like any entity, but ...
    - Has at least one *special relationship* – ***contains***.
- Operations, minimally CRUD, that manipulate entity types and instances:
  - Create
  - Retrieve
  - Update
  - Delete
  - Reference/Identify/... ...

## What is REST architecture?

REST stands for REpresentational State Transfer. REST is web standards based architecture and uses HTTP Protocol. It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods. REST was first introduced by Roy Fielding in 2000.

In REST architecture, a REST Server simply provides access to resources and REST client accesses and modifies the resources. Here each resource is identified by URIs/ global IDs. REST uses various representation to represent a resource like text, JSON, XML. JSON is the most popular one.

## HTTP methods

Following four HTTP methods are commonly used in REST based architecture.

- **GET** – Provides a read only access to a resource.
- **POST** – Used to create a new resource.
- **DELETE** – Used to remove a resource.
- **PUT** – Used to update a existing resource or create a new resource.

## Introduction to RESTful web services

A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. This interoperability (e.g., between Java and Python, or Windows and Linux applications) is due to the use of open standards.

Web services based on REST Architecture are known as RESTful web services. These webservices uses HTTP methods to implement the concept of REST architecture. A RESTful web service usually defines a URI, Uniform Resource Identifier a service, provides resource representation such as JSON and set of HTTP Methods.

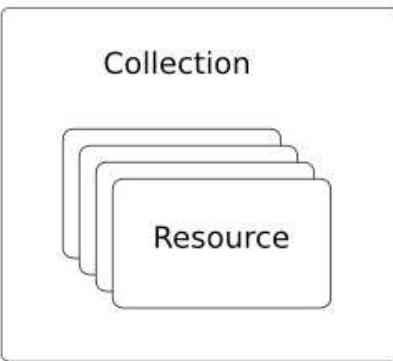
## Creating RESTful Webservice

In next chapters, we'll create a webservice say user management with following functionalities –

| Sr.No. | URI                     | HTTP Method | POST body   | Result                      |
|--------|-------------------------|-------------|-------------|-----------------------------|
| 1      | /UserService/users      | GET         | empty       | Show list of all the users. |
| 2      | /UserService/addUser    | POST        | JSON String | Add details of new user.    |
| 3      | /UserService/getUser/id | GET         | empty       | Show details of a user.     |

# REST and Resources

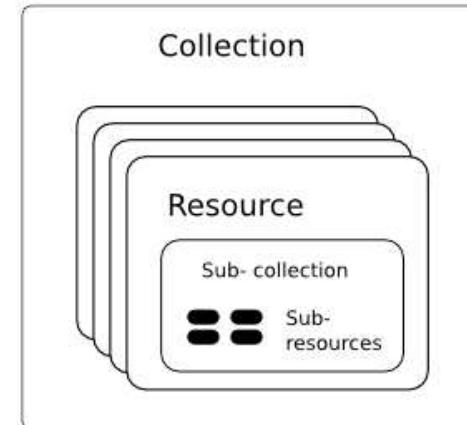
## Resource Model



A Collection with  
Resources

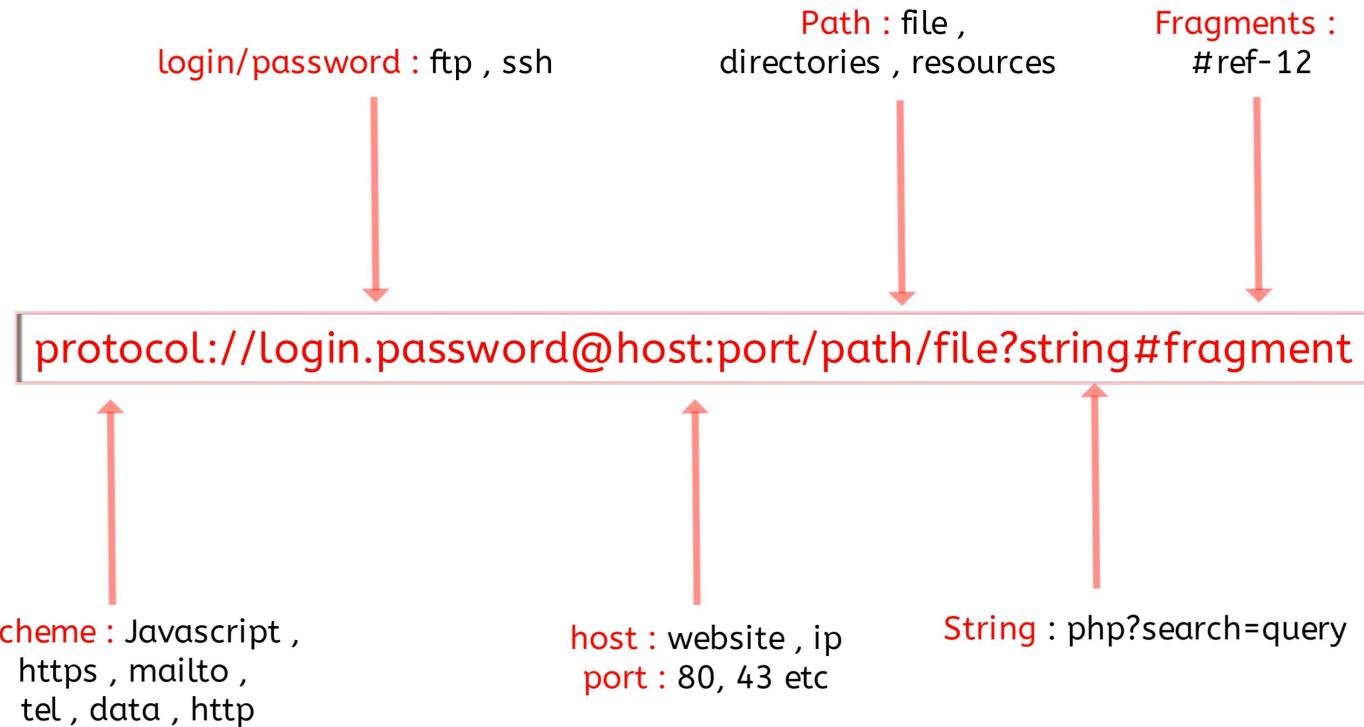


A Singleton  
Resource



Sub-collections and  
Sub-resources

# URLs



`jdbc:mysql://columbia-examples.ckkqqktwkcji.us-east-1.rds.amazonaws.com:3306`

# Simplistic, Conceptual Mapping (Examples)

| REST Method | Resource Path             | Relational Operation                                                                                             | DB Resource                |
|-------------|---------------------------|------------------------------------------------------------------------------------------------------------------|----------------------------|
| DELETE      | /people                   | DROP TABLE                                                                                                       | people table               |
| POST        | /people                   | INSERT INTO PEOPLE (...) VALUES(...)                                                                             | people table<br>people row |
| GET         | /people/21                | SHOW KEYS FROM people ...;<br><br>SELECT * FROM people WHERE<br>playerID= 21                                     | people row                 |
| GET         | /people/21/batting        | SELECT batting.* FROM<br>people JOIN batting USING(playerID)<br>WHERE playerID=21                                |                            |
| GET         | /people/21/batting/2004_1 | SELECT batting.* FROM<br>people JOIN batting USING(playerID)<br>WHERE playerID=21<br>AND yearID=2004 AND stint=1 |                            |

# Application Architecture

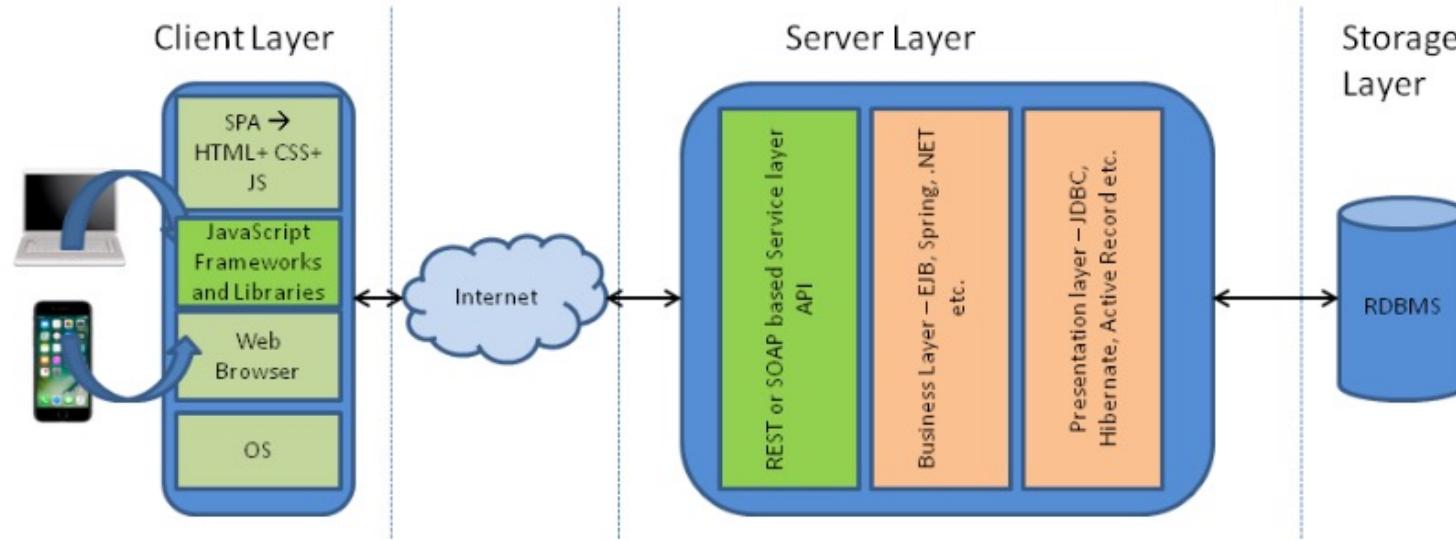


Diagram 2: The moving of the Web Layer from the Server to the Client

# Walkthrough

- Simple web application template.
- Calling some cloud APIs.

# *Sample Projects*

# Projects

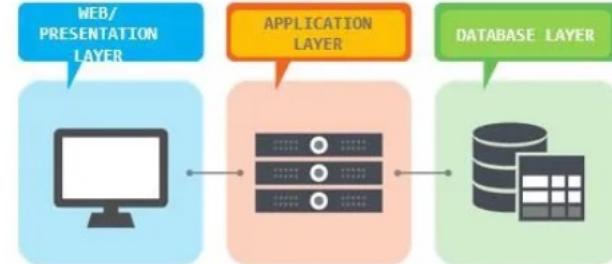
- The programming track will implement a simple, full stack web application.

## Full-stack Web Developer

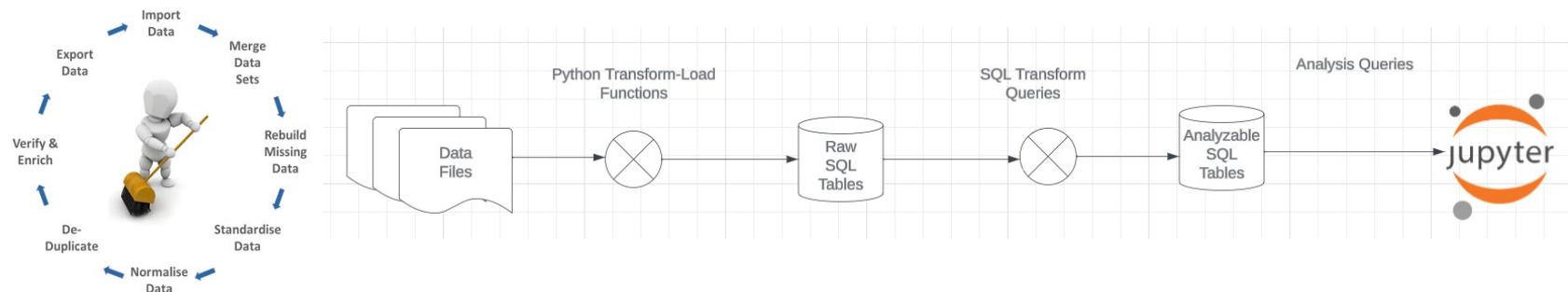
A full-stack web developer is a person who can develop both **client** and **server** software.

In addition to mastering HTML and CSS, he/she also knows how to:

- Program a **browser** (e.g. using JavaScript, jQuery, Angular, or Vue)
- Program a **server** (e.g. using PHP, ASP, Python, or Node)
- Program a **database** (e.g. using SQL, SQLite, or MongoDB)



- The non-programming track will implement a data engineering and visualization process in a Jupyter notebook.



# Game of Thrones

- Bottom-Up Data Mapping:
  - “Nouns” usually map to Entity/Entity Set.
  - Nouns inside other nouns often map to:
    - Attribute
    - Relationship
  - Verbs often map to relationships.
  - Adjectives usually map to properties.
- We will start with a subset of the information:
  - Game of Thrones:
    - Episodes
    - Characters
  - IMDB:
    - names\_basics
    - title\_basics
- Entities Sets
  - Character
  - Season
  - Episode
  - Scene
  - Location, Sublocation
  - ... ...
- Relationships
  - Character – Scene
  - Character – Character (e.g. KilledBy)
  - Season – IMDB Title
  - Character – IMDB Name
  - ... ...

# Game of Thrones

- Bottom-Up Data Mapping:

- IMDB: <https://developer.imdb.com/non-commercial-datasets/>

- Do not download.
    - Despite being “tiny” compared to the real world.
    - The datasets are too big for most laptops.

- We have info

- Game of Thrones: <https://github.com/jeffreylancaster/game-of-thrones>

- IMDB datasets
    - names\_basics
    - title\_basics

- Entities Sets

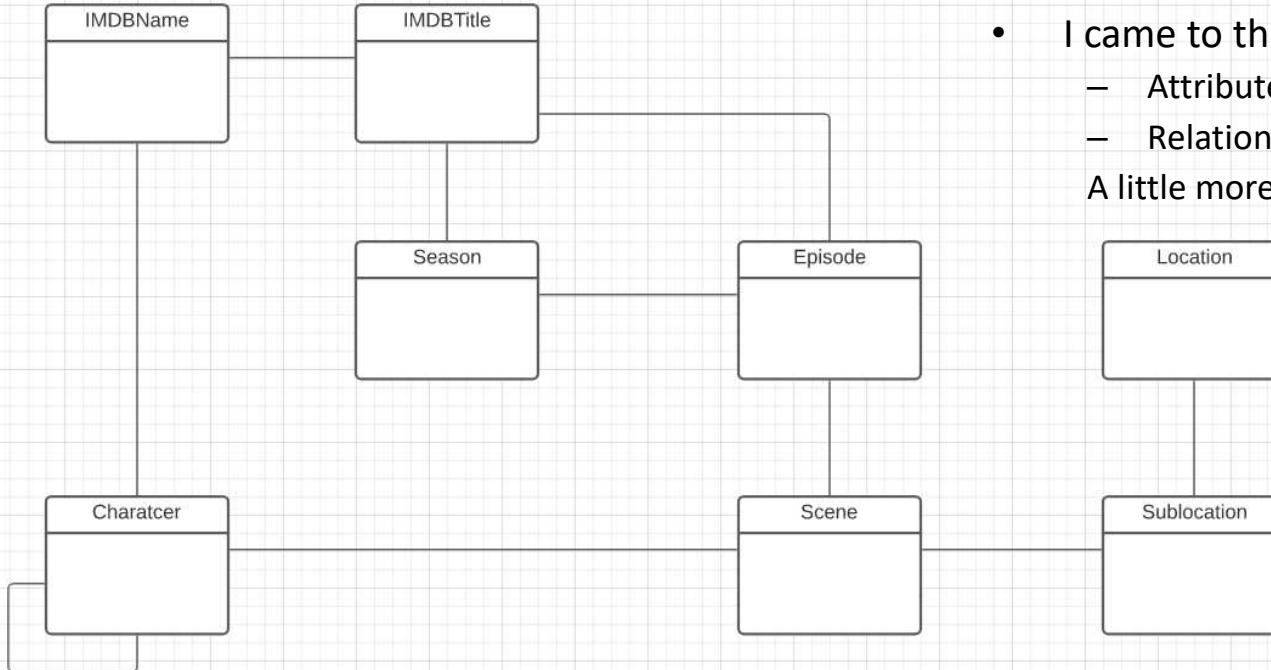
Character – IMDB Name

... ...

# Game of Thrones – Conceptual Model

Add shapes in Lucidchart ...

Add Entity Relationship Shapes



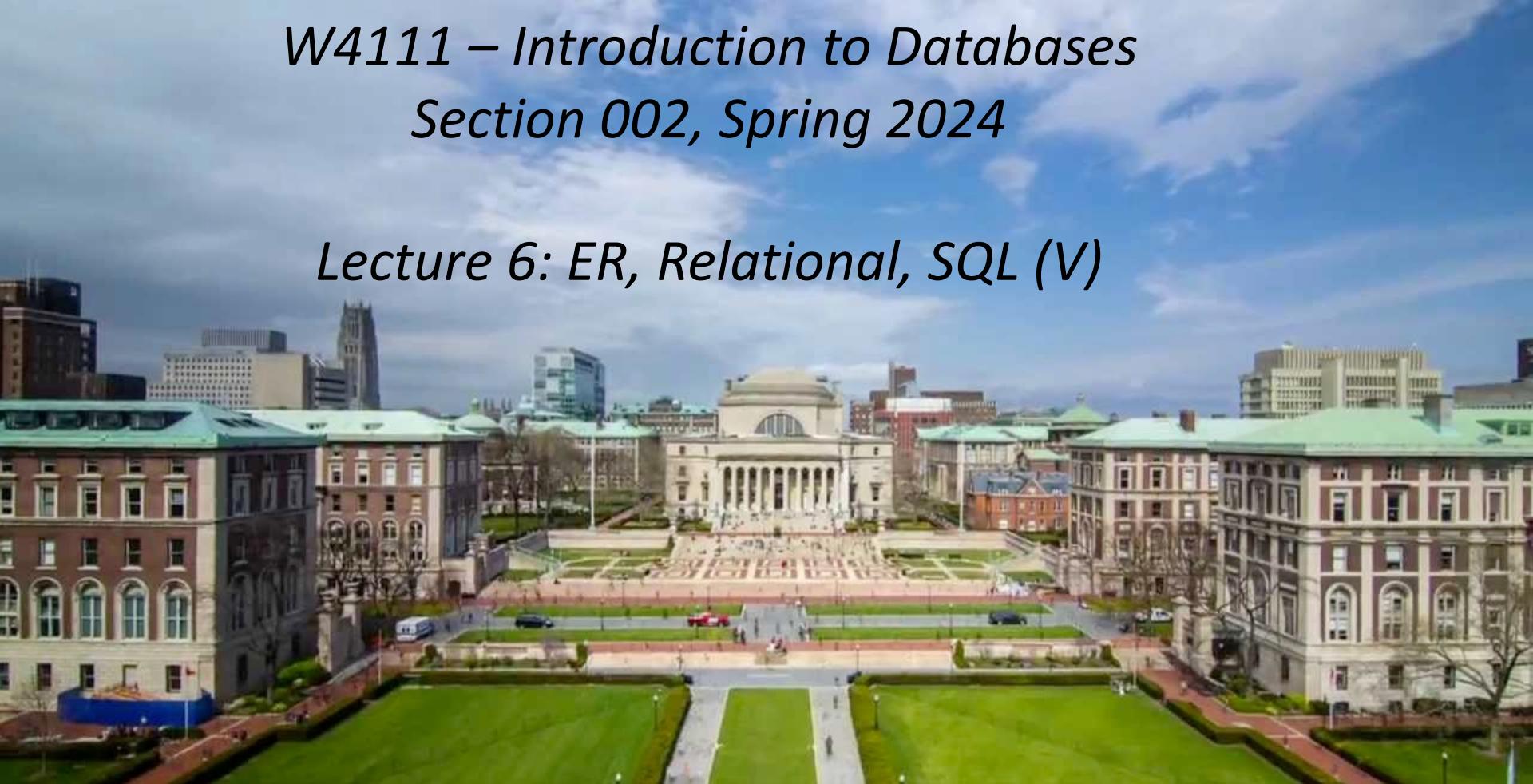
- With a little
  - Data exploration
  - Common sense
  - Judgment/experience
- I came to this conceptual model.
  - Attributes unspecified
  - Relationship required/cardinality unspecified.A little more exploration is needed.

# Sample Projects

- Walkthrough of Web Apps:
  - /Users/donaldferguson/Dropbox/000/000-A-Current-Examples/W4111-FastAPI-IMDB-Solution
  - /Users/donaldferguson/Dropbox/000/000-A-Current-Examples/current-dashboard
- Data Engineering:
  - /Users/donaldferguson/Dropbox/000/000-Columbia/2024-Spring/W4111-Intro-to-Databases-Spring-2024/Lectures/s-2024-Lecture-3/More-Data-Engineering.ipynb
  - And others.

*W4111 – Introduction to Databases  
Section 002, Spring 2024*

*Lecture 6: ER, Relational, SQL (V)*



*W4111 – Introduction to Databases  
Section 002, Spring 2024*

*Lecture 6: ER, Relational, SQL (V)*

We will start in a couple of minutes.

# *Contents*

# Contents

- Some Observations
- ER (Diagram) Modeling – Aggregation, other notations
- SQL Continued (Chapter 3, 4, 5)
- Applications and Databases
- Project Examples:
  - Web Applications and REST
  - Data Engineering and Visualization

# *Some Observations*

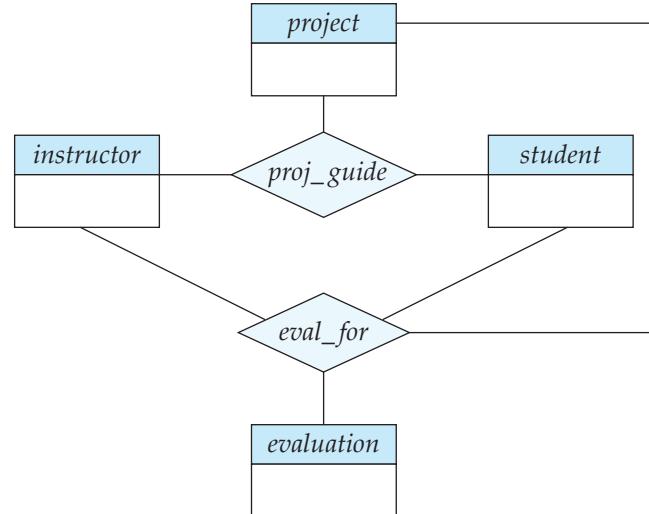
# Database design, Entity-Relationship Model Some Complex Examples

# *Aggregation*



# Aggregation

- Consider the ternary relationship *proj\_guide*, which we saw earlier
- Suppose we want to record evaluations of a student by a guide on a project





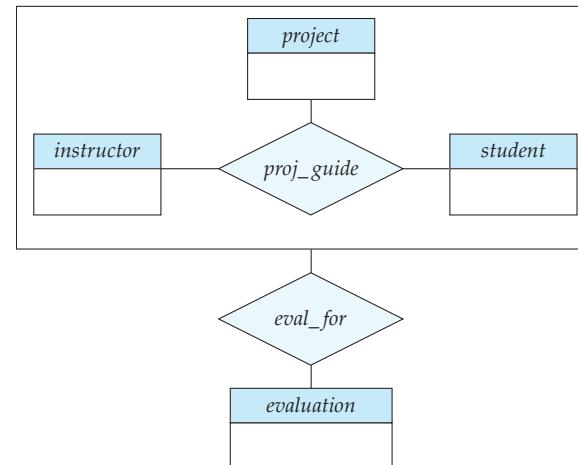
# Aggregation (Cont.)

- Relationship sets *eval\_for* and *proj\_guide* represent overlapping information
  - Every *eval\_for* relationship corresponds to a *proj\_guide* relationship
  - However, some *proj\_guide* relationships may not correspond to any *eval\_for* relationships
    - So we can't discard the *proj\_guide* relationship
- Eliminate this redundancy via *aggregation*
  - Treat relationship as an abstract entity
  - Allows relationships between relationships
  - Abstraction of relationship into new entity



# Aggregation (Cont.)

- Eliminate this redundancy via *aggregation* without introducing redundancy, the following diagram represents:
  - A student is guided by a particular instructor on a particular project
  - A student, instructor, project combination may have an associated evaluation



The simplest way to handle in relational is an associative entity.

Some thoughts here:

<https://www.geeksforgeeks.org/aggregate-data-model-in-nosql/>



# Reduction to Relational Schemas

- To represent aggregation, create a schema containing
  - Primary key of the aggregated relationship,
  - The primary key of the associated entity set
  - Any descriptive attributes
- In our example:
  - The schema *eval\_for* is:  
$$\text{eval\_for} (s\_ID, project\_id, i\_ID, evaluation\_id)$$
  - The schema *proj\_guide* is redundant.

## *Other Notations*

*Note: Make sure you show ER vs. UML Class Diagrams*



# Summary of Symbols Used in E-R Notation



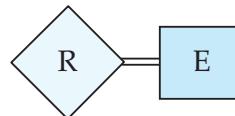
entity set



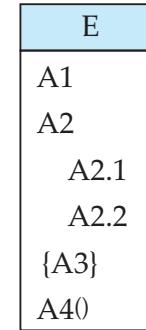
relationship set



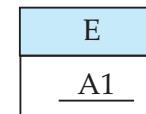
identifying  
relationship set  
for weak entity set



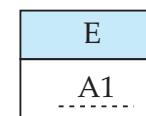
total participation  
of entity set in  
relationship



attributes:  
simple (A1),  
composite (A2) and  
multivalued (A3)  
derived (A4)



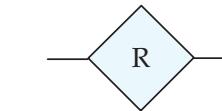
primary key



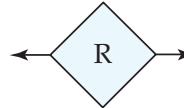
discriminating  
attribute of  
weak entity set



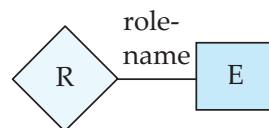
# Symbols Used in E-R Notation (Cont.)



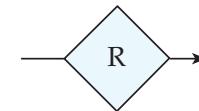
many-to-many  
relationship



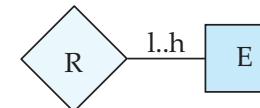
one-to-one  
relationship



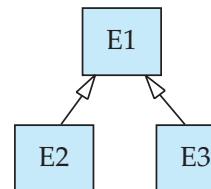
role  
indicator



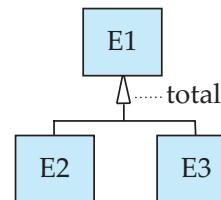
many-to-one  
relationship



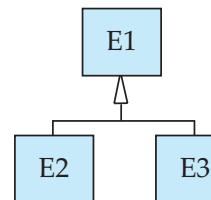
cardinality  
limits



ISA: generalization  
or specialization



total (disjoint)  
generalization



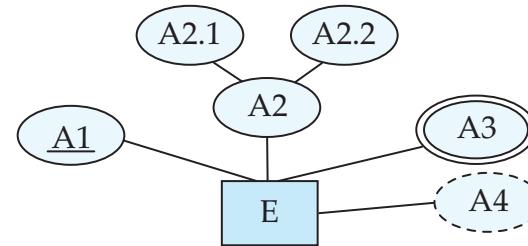
disjoint  
generalization



# Alternative ER Notations

- Chen, IDE1FX, ...

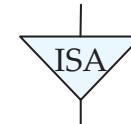
entity set E with  
simple attribute A1,  
composite attribute A2,  
multivalued attribute A3,  
derived attribute A4,  
and primary key A1



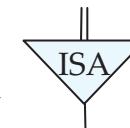
weak entity set



generalization



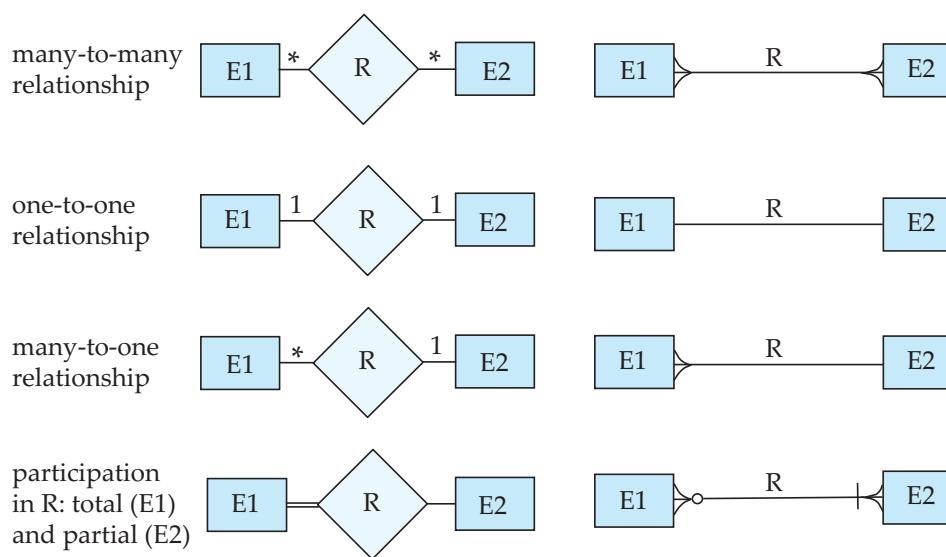
total  
generalization





# Alternative ER Notations

Chen



IDE1FX (Crows feet notation)



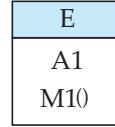
# UML

- **UML**: Unified Modeling Language
- UML has many components to graphically model different aspects of an entire software system
- UML Class Diagrams correspond to E-R Diagram, but several differences.

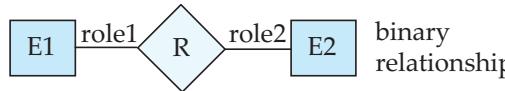


# ER vs. UML Class Diagrams

## ER Diagram Notation

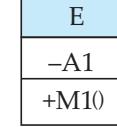


entity with  
attributes (simple,  
composite,  
multivalued, derived)

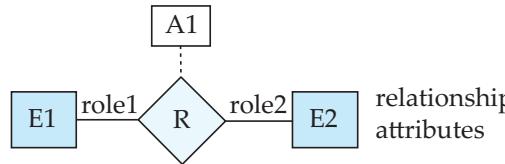


binary  
relationship

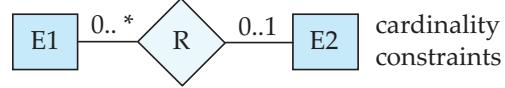
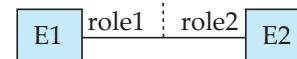
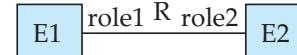
## Equivalent in UML



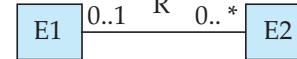
class with simple attributes  
and methods (attribute  
prefixes: + = public,  
- = private, # = protected)



relationship  
attributes



cardinality  
constraints

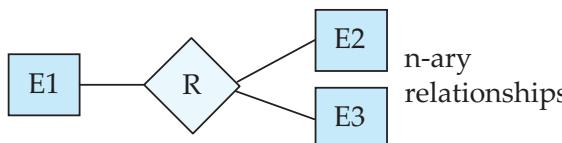


- \* Note reversal of position in cardinality constraint depiction

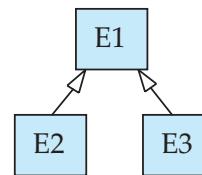


# ER vs. UML Class Diagrams

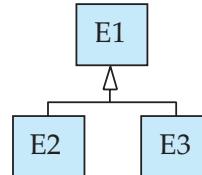
## ER Diagram Notation



n-ary relationships

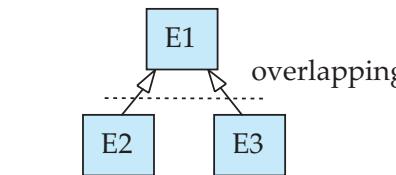
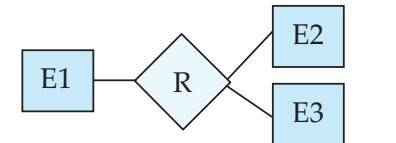


overlapping  
generalization

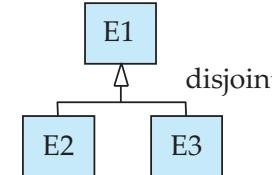


disjoint  
generalization

## Equivalent in UML



overlapping



disjoint

- \* Generalization can use merged or separate arrows independent of disjoint/overlapping

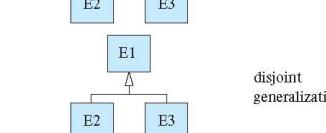
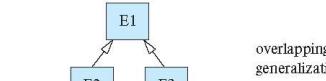
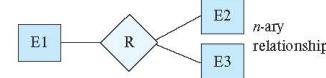
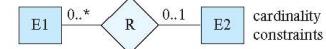
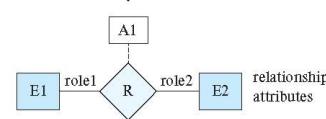
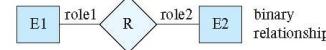


# ER vs. UML Class Diagrams

ER Diagram Notation



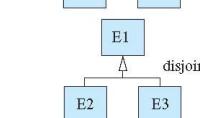
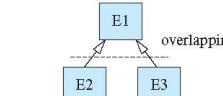
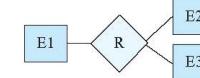
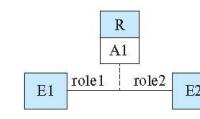
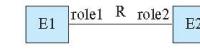
entity with  
attributes (simple,  
composite,  
multivalued, derived)



Equivalent in UML



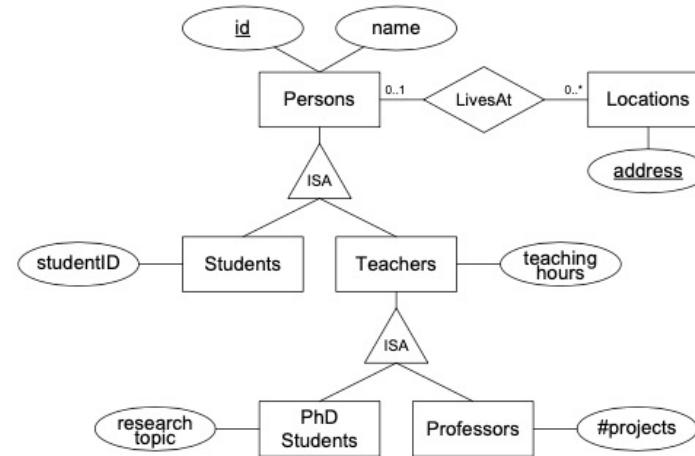
class with simple attributes  
and methods (attribute  
prefixes: + = public,  
- = private, # = protected)



# ISA Relationship



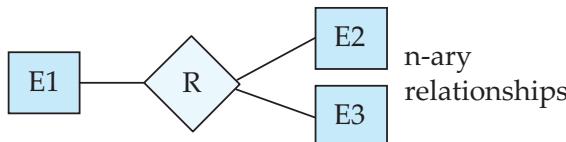
## ISA Relationship





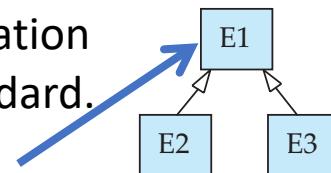
# ER vs. UML Class Diagrams

## ER Diagram Notation

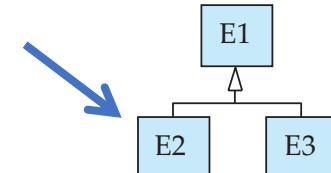


n-ary relationships

I use this approach  
in Crow's Foot Notation  
but that is not standard.

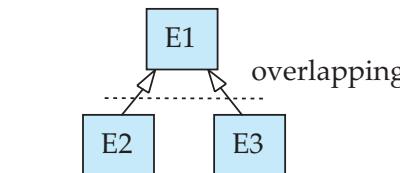
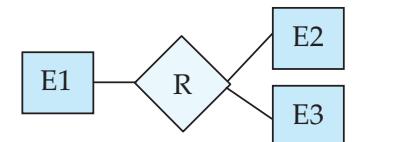


overlapping  
generalization

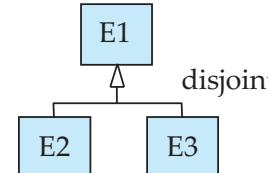


disjoint  
generalization

## Equivalent in UML



overlapping



disjoint

- \* Generalization can use merged or separate arrows independent of disjoint/overlapping

# *SQL*

# *More on Codd's Rules*

# Codd's 12 Rules

## Rule 1: Information Rule

The data stored in a database, may it be user data or metadata, must be a value of some table cell. Everything in a database must be stored in a table format.

## Rule 2: Guaranteed Access Rule

Every single data element (value) is guaranteed to be accessible logically with a combination of table-name, primary-key (row value), and attribute-name (column value). No other means, such as pointers, can be used to access data.

## Rule 3: Systematic Treatment of NULL Values

The NULL values in a database must be given a systematic and uniform treatment. This is a very important rule because a NULL can be interpreted as one the following – data is missing, data is not known, or data is not applicable.

## Rule 4: Active Online Catalog

The structure description of the entire database must be stored in an online catalog, known as data dictionary, which can be accessed by authorized users. Users can use the same query language to access the catalog which they use to access the database itself.

## Rule 5: Comprehensive Data Sub-Language Rule

A database can only be accessed using a language having linear syntax that supports data definition, data manipulation, and transaction management operations. This language can be used directly or by means of some application. If the database allows access to data without any help of this language, then it is considered as a violation.

## Rule 6: View Updating Rule

All the views of a database, which can theoretically be updated, must also be updatable by the system.

# Codd's 12 Rules

## Rule 7: High-Level Insert, Update, and Delete Rule

A database must support high-level insertion, updation, and deletion. This must not be limited to a single row, that is, it must also support union, intersection and minus operations to yield sets of data records.

## Rule 8: Physical Data Independence

**The data stored in a database must be independent of the applications that access the database. Any change in the physical structure of a database must not have any impact on how the data is being accessed by external applications.**

## Rule 9: Logical Data Independence

**The logical data in a database must be independent of its user's view (application). Any change in logical data must not affect the applications using it. For example, if two tables are merged or one is split into two different tables, there should be no impact or change on the user application. This is one of the most difficult rule to apply.**

## Rule 10: Integrity Independence

**A database must be independent of the application that uses it. All its integrity constraints can be independently modified without the need of any change in the application. This rule makes a database independent of the front-end application and its interface.**

## Rule 11: Distribution Independence

The end-user must not be able to see that the data is distributed over various locations. Users should always get the impression that the data is located at one site only. This rule has been regarded as the foundation of distributed database systems.

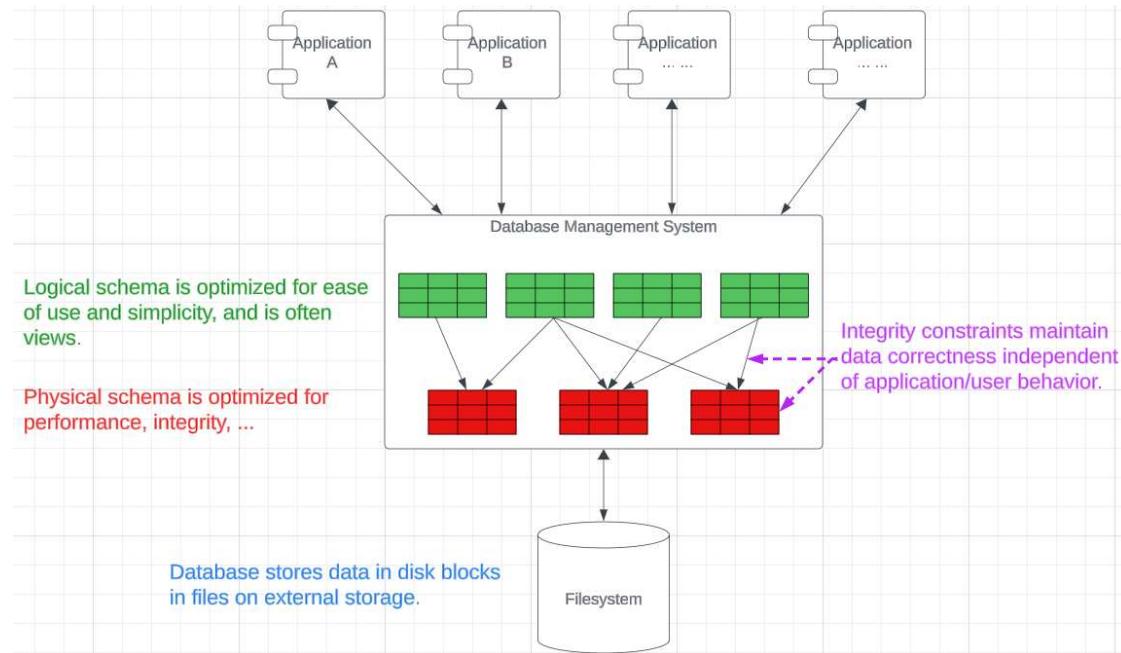
## Rule 12: Non-Subversion Rule

If a system has an interface that provides access to low-level records, then the interface must not be able to subvert the system and bypass security and integrity constraints.

# Physical Data, Logical Data, Integrity Independence

Overly simplistic explanation:

- Logical Independence:
  - “Uses views”
  - Adapts a potentially complex model to how a user “thinks” about the data.
  - Enables changing the optimized physical model without affecting users – reimplement views.
- Physical Independence:
  - The physical schema is optimized for performance, complex integrity, ...
  - Changes as the application evolves, performance requirements change, etc.
  - The representation in the database is independent of the physical storage model.
- Integrity Independence:
  - Integrity constraints associated with the database tables/entities partially allows modification without requiring application changes.
  - Integrity does not rely on correctness of multiple applications and users.



# *Some Types and Functions*

# Functions

## MySQL CHEAT SHEET: STRING FUNCTIONS

by [sqlbackupandftp.com](http://sqlbackupandftp.com) with ♥

### MEASUREMENT

Return a string containing binary representation of a number

`BIN (12) = '1100'`

Return length of argument in bits

`BIT_LENGTH ('MySQL') = 40`

Return number of characters in argument

`CHAR_LENGTH ('MySQL') = 5`

`CHARACTER_LENGTH ('MySQL') = 5`

Return the length of a string in bytes

`LENGTH ('Ø') = 2`

`LENGTH ('A') = 1`

`OCTET_LENGTH ('Ø') = 2`

`OCTET_LENGTH ('X') = 1`

Return a soundex string

`SOUNDEX ('MySQL') = 'M240'`

`SOUNDEX ('MySQLDatabase') = 'M24312'`

Compare two strings

`STRCMP ('A', 'A') = 0`

`STRCMP ('A', 'B') = -1`

`STRCMP ('B', 'A') = 1`

### SEARCH

Return the index of the first occurrence of substring

`INSTR ('MySQL', 'Sql') = 3`

`INSTR ('Sql', 'MySQL') = 0`

Return the position of the first occurrence of substring

`LOCATE ('Sql', 'MySQLSql') = 3`

`LOCATE ('xSql', 'MySQL') = 0`

`LOCATE ('Sql', 'MySQLSql', 5) = 6`

`POSITION('Sql' IN 'MySQLSql') = 3`

Pattern matching using regular expressions

`'abc' RLIKE '[a-z]+ = 1`

`'123' RLIKE '[a-z]+ = 0`

Return a substring from a string before the specified number of occurrences of the delimiter

`SUBSTRING_INDEX ('A:B:C', ':', 1) = 'A'`

`SUBSTRING_INDEX ('A:B:C', ':', 2) = 'A:B'`

`SUBSTRING_INDEX ('A:B:C', ':', -2) = 'B:C'`

### CONVERSION

Return numeric value of left-most character

`ASCII ('2') = 50`

`ASCII (2) = 50`

`ASCII ('ð') = 100`

Return the character for each number passed

`CHAR (45*256+45) = CHAR (45,45) = '-'`

`CHARSET(CHAR (X'65' USING utf8)) = 'utf8'`

Decode to / from a base-64 string

`TO_BASE64 ('abc') = 'YmJz'`

`FROM_BASE64 ('YmJz') = 'abc'`

Convert string or number to its hexadecimal representation

`X'616263' = 'abc'`

`HEX ('abc') = 616263`

`HEX(255) = 'FF'`

`CONV(HEX(255), 16, 10) = 255`

Convert each pair of hexadecimal digits to a character

`UNHEX ('4D7953514C') = 'MySQL'`

`UNHEX ('GG') = NULL`

`UNHEX (HEX ('abc')) = 'abc'`

Return the argument in lowercase

`LOWER ('MySQL') = 'mysql'`

`LCASE ('MySQL') = 'mysql'`

Load the named file

`SET blob_col=LOAD_FILE ('/tmp/picture')`

Return a string containing octal representation of a number

`OCT (12) = '14'`

Return character code for leftmost character of the argument

`ORD ('2') = 50`

Escape the argument for use in an SQL statement

`QUOTE ('Don\'t!') = 'Don\'t!'`

`QUOTE (NULL) = NULL`

Convert to uppercase

`UPPER ('mysql') = 'MYSQL'`

`UCASE ('mysql') = 'MYSQL'`

### MODIFICATION

Return concatenated string

`CONCAT ('My', ' ', 'SQL') = 'MySQL'`

`CONCAT ('My', NULL, 'SQL') = NULL`

`CONCAT ('14.3') = '14.3'`

Return concatenate with separator

`CONCAT_WS (' ', 'My', 'Sql') = 'My,Sql'`

`CONCAT_WS (' ', 'My', NULL, 'Sql') = 'My,Sql'`

Return a number formatted to specified number of decimal places

`FORMAT ('12332.123456, 4) = 12,332.1235`

`FORMAT ('12332., 4) = 12,332.1000`

`FORMAT ('12332.2, 0) = 12332.2`

`FORMAT ('12332.2, 2, 'de_DE') = 12.332,20`

Insert a substring at the specified position up to the specified number of characters

`INSERT ('12345', 3, 2, 'ABC') = '12ABC5'`

`INSERT ('12345', 10, 2, 'ABC') = '12345'`

`INSERT ('12345', 3, 10, 'ABC') = '12ABC'`

Return the leftmost number of characters as specified

`LEFT ('MySQL', 2) = 'My'`

Return the string argument, left-padded with the specified string

`LPAD ('Sql', 2, ':') = 'S:q'`

`LPAD ('Sql', 4, ':') = 'S:ql'`

`LPAD ('Sql', 7, ':') = ':):)Sql'`

Remove leading spaces

`LTRIM (' ' MySQL) = 'MySQL'`

Repeat a string the specified number of times

`REPEAT ('MySQL', 3) = 'MySQLMySQLMySQL'`

Replace occurrences of a specified string

`REPLACE ('NoSql', 'No', 'My') = 'MySql'`

Reverse the characters in a string

`REVERSE ('MySQL') = 'lqSytM'`

Return the specified rightmost number of characters

`RIGHT ('MySQL', 3) = 'Sql'`

Return the string argument, right-padded with the specified string

`RPAD ('Sql', 2, ':') = 'Sql'`

`RPAD ('Sql', 4, ':') = 'Sql: '`

`RPAD ('Sql', 7, ':') = 'Sql:;;'`

Remove trailing spaces

`RTRIM ('MySQL ') = 'MySQL'`

Return a string of the specified number of spaces

`SPACE ('6') = ''`

Return the substring as specified

`SUBSTRING=SUBSTR-MID ('MySQL',3) = 'Sql'`

`SUBSTRING=SUBSTR-MID ('MySQL', FROM 4) = 'ql'`

`SUBSTRING=SUBSTR-MID ('MySQL',3,1) = 'S'`

`SUBSTRING=SUBSTR-MID ('MySQL',-3) = 'ol'`

`SUBSTRING=SUBSTR-MID ('MySQL', FROM -4 FOR 2) = 'yS'`

Remove leading and trailing spaces

`TRIM(' MySql ') = ' MySql '`

`TRIM(LEADING 'x' FROM 'xxxSqlMy') = 'MySql'`

`TRIM(BOTH 'My' FROM 'MySqlMy') = 'Sql'`

`TRIM(TRAILING 'Sql' FROM 'MySql') = 'My'`

### SETS

Return string at index number

`ELT (1, 'ej', 'Heja', 'hej', 'foo') = 'ej'`

`ELT (4, 'ej', 'Heja', 'hej', 'foo') = 'foo'`

Return a string such that for every bit set in the value bits, you get an on string and for every unset bit, you get an off string

`EXPORT_SET (5,'Y','N','Y','N',4) = 'Y,N,Y,N'`

`EXPORT_SET (6,'1','0','1','0',6) = '0,1,1,0,0,0'`

Return the index (position) of the first argument in the subsequent arguments

`FIELD ('ej','Hj','ej','Heja','hej','oo') = 2`

`FIELD ('fo','Hj','ej','Heja','hej','oo') = 0`

Return the index position of the first argument within the second argument

`FIND_IN_SET ('b', 'a,b,c,d') = 2`

`FIND_IN_SET ('z', 'a,b,c,d') = 0`

`FIND_IN_SET ('a', 'a,b,c,d') = 0`

Return a set of comma-separated strings that have the corresponding bit in bits set

`MAKE_SET (1,'a','b','c') = 'a'`

`MAKE_SET (1|4,'ab','cd','ef') = 'ab,ef'`

`MAKE_SET (1|4,'ab','cd',NULL,'ef') = 'ab'`

`MAKE_SET (0,'a','b','c') = ''`

### TEXT FUNCTIONS

#### CONCATENATION

Use the || operator to concatenate two strings:  
`SELECT 'Hi ' || 'there!';`  
-- result: Hi there!

Remember that you can concatenate only character strings using |||. Use this trick for numbers:  
`SELECT '' || 4 || 2;`  
-- result: 42

Some databases implement non-standard solutions for concatenating strings like CONCAT() or CONCAT\_WS(). Check the documentation for your specific database.

#### LIKE OPERATOR – PATTERN MATCHING

Use the \_ character to replace any single character. Use the % character to replace any number of characters (including 0 characters).

Fetch all names that start with any letter followed by 'atherine':  
`SELECT name  
FROM names  
WHERE name LIKE '_atherine';`

Fetch all names that end with 'a':  
`SELECT name  
FROM names  
WHERE name LIKE '%a';`

#### USEFUL FUNCTIONS

Get the count of characters in a string:  
`SELECT LENGTH('LearnSQL.com');`  
-- result: 12

Convert all letters to lowercase:  
`SELECT LOWER('LEARNSQL.COM');`  
-- result: learnsql.com

Convert all letters to uppercase:  
`SELECT UPPER('LearnSQL.com');`  
-- result: LEARNSQL.COM

Convert all letters to lowercase and all first letters to uppercase (not implemented in MySQL and SQL Server):  
`SELECT INITCAP('edgar frank ted codd');`  
-- result: Edgar Frank Ted Codd

Get just a part of a string:  
`SELECT SUBSTRING('LearnSQL.com', 9);`  
-- result: .com

Select a part of a string:  
`SELECT REPLACE('LearnSQL.com', 'SQL', 'Python');`  
-- result: LearnPython.com

### NUMERIC FUNCTIONS

#### BASIC OPERATIONS

Use +, -, \*, / to do some basic math. To get the number of seconds in a week:  
`SELECT 60 * 60 * 24 * 7;`  
-- result: 604800

#### CASTING

From time to time, you need to change the type of a number. The CAST() function is there to help you out. It lets you change the type of almost anything (integer, numeric, double precision, varchar, and many more).

Get the number as an integer (without rounding):  
`SELECT CAST(1234.567 AS integer);`  
-- result: 1234

Change a column type to double precision  
`SELECT CAST(column AS double precision);`

#### USEFUL FUNCTIONS

Get the remainder of a division:  
`SELECT MOD(13, 2);`  
-- result: 1

Round a number to its nearest integer:  
`SELECT ROUND(1234.56789);`  
-- result: 1235

Round a number to three decimal places:  
`SELECT ROUND(1234.56789, 3);`  
-- result: 1234.568

PostgreSQL requires the first argument to be of the type numeric – cast the number when needed.

To round the number up:  
`SELECT CEIL(13.1);`  
-- result: 14  
`SELECT CEIL(-13.9);`  
-- result: -13  
The CEIL(x) function returns the **smallest** integer **not less than** x. In SQL Server, the function is called CEILING().

To round the number down:  
`SELECT FLOOR(13.8);`  
-- result: 13  
`SELECT FLOOR(-13.2);`  
-- result: -14  
The FLOOR(x) function returns the **greatest** integer **not greater than** x.

To round towards 0 irrespective of the sign of a number:  
`SELECT TRUNC(13.5);`  
-- result: 13  
`SELECT TRUNC(-13.5);`  
-- result: -13  
TRUNC(x) works the same way as CAST(x AS integer). In MySQL, the function is called TRUNCATE().

To get the absolute value of a number:  
`SELECT ABS(-12);`  
-- result: 12

To get the square root of a number:  
`SELECT SQRT(9);`  
-- result: 3

### NULLS

To retrieve all rows with a missing value in the price column:  
`WHERE price IS NULL`

To retrieve all rows with the weight column populated:  
`WHERE weight IS NOT NULL`

Why shouldn't you use price = NULL or weight != NULL?  
Because databases don't know if those expressions are true or false – they are evaluated as NULLs.

Moreover, if you use a function or concatenation on a column that is NULL in some rows, then it will get propagated. Take a look:

| domain          | LENGTH(domain) |
|-----------------|----------------|
| LearnSQL.com    | 12             |
| LearnPython.com | 15             |
| NULL            | NULL           |
| vertabelo.com   | 13             |

#### USEFUL FUNCTIONS

COALESCE(x, y, ...)  
To replace NULL in a query with something meaningful:

```
SELECT
 domain,
 COALESCE(domain, 'domain missing')
FROM contacts;
```

| domain       | coalesce       |
|--------------|----------------|
| LearnSQL.com | LearnSQL.com   |
| NULL         | domain missing |

The COALESCE() function takes any number of arguments and returns the value of the first argument that isn't NULL.

NULIF(x, y)  
To save yourself from division by 0 errors:

```
SELECT
 last_month,
 this_month,
 this_month * 100.0
 / NULIF(last_month, 0)
 AS better_by_percent
FROM video_views;
```

| last_month | this_month | better_by_percent |
|------------|------------|-------------------|
| 723786     | 1085679    | 150.0             |
| 0          | 178123     | NULL              |

The NULIF(x, y) function will return NULL if x is the same as y, else it will return the x value.

### CASE WHEN

The basic version of CASE WHEN checks if the values are equal (e.g., if fee is equal to 50, then 'normal' is returned). If there isn't a matching value in the CASE WHEN, then the ELSE value will be returned (e.g., if fee is equal to 49, then 'not available' will show up).

```
SELECT
CASE fee
 WHEN 50 THEN 'normal'
 WHEN 10 THEN 'reduced'
 WHEN 0 THEN 'free'
 ELSE 'not available'
END AS tariff
FROM ticket_types;
```

The most popular type is the **searched CASE WHEN** – it lets you pass conditions (as you'd write them in the WHERE clause), evaluates them in order, then returns the value for the first condition met.

```
SELECT
CASE
 WHEN score >= 90 THEN 'A'
 WHEN score > 60 THEN 'B'
 ELSE 'F'
END AS grade
FROM test_results;
```

Here, all students who scored at least 90 will get an A, those with the score above 60 (and below 90) will get a B, and the rest will receive an F.

### TROUBLESHOOTING

#### Integer division

When you don't see the decimal places you expect, it means that you are dividing between two integers. Cast one to decimal:  
`CAST(123 AS decimal) / 2`

#### Division by 0

To avoid this error, make sure that the denominator is not equal to 0. You can use the NULIF() function to replace 0 with a NULL, which will result in a NULL for the whole expression:  
`count / NULIF(count_all, 0)`

#### Inexact calculations

If you do calculations using real (floating point) numbers, you'll end up with some inaccuracies. This is because this type is meant for scientific calculations such as calculating the velocity. Whenever you need accuracy (such as dealing with monetary values), use the decimal/numeric type (or money if available).

#### Errors when rounding with a specified precision

Most databases won't complain, but do check the documentation if they do. For example, if you want to specify the rounding precision in PostgreSQL, the value must be of the numeric type.

Try out the interactive **Standard SQL Functions** course at [LearnSQL.com](https://LearnSQL.com), and check out our other SQL courses.

LearnSQL.com is owned by Vertabelo SA  
vertabelo.com | CC-BY-NC-ND Vertabelo SA

# Functions

## Standard SQL Functions Cheat Sheet

### AGGREGATION AND GROUPING

- `COUNT(expr)` – the count of values for the rows within the group
- `SUM(expr)` – the sum of values within the group
- `AVG(expr)` – the average value for the rows within the group
- `MIN(expr)` – the minimum value within the group
- `MAX(expr)` – the maximum value within the group

To get the number of rows in the table:

```
SELECT COUNT(*)
FROM city;
```

To get the number of non-NULL values in a column:

```
SELECT COUNT(rating)
FROM city;
```

To get the count of unique values in a column:

```
SELECT COUNT(DISTINCT country_id)
FROM city;
```

### GROUP BY

| CITY      |            |
|-----------|------------|
| name      | country_id |
| Paris     | 1          |
| Marseille | 1          |
| Lyon      | 1          |
| Berlin    | 2          |
| Hamburg   | 2          |
| Munich    | 2          |
| Warsaw    | 4          |
| Cracow    | 4          |

→

| CITY       |       |
|------------|-------|
| country_id | count |
| 1          | 3     |
| 2          | 3     |
| 4          | 2     |

The example above – the count of cities in each country:

```
SELECT name, COUNT(country_id)
FROM city
GROUP BY name;
```

The average rating for the city:

```
SELECT city_id, AVG(rating)
FROM ratings
GROUP BY city_id;
```

### Common mistake: COUNT(\*) and LEFT JOIN

When you join the tables like this: `client LEFT JOIN project`, and you want to get the number of projects for every client you know, `COUNT(*)` will return 1 for each client even if you've never worked for them. This is because, they're still present in the list but with the NULL in the fields related to the project after the JOIN. To get the correct count (0 for the clients you've never worked for), count the values in a column of the other table, e.g., `COUNT(project_name)`. Check out this [exercise](#) to see an example.

### DATE AND TIME

There are 3 main time-related types: `date`, `time`, and `timestamp`. Time is expressed using a 24-hour clock, and it can be as vague as just hour and minutes (e.g., 15:30 – 3:30 p.m.) or as precise as microseconds and time zone (as shown below):

2021-12-31 14:39:53.662522-05



YYYY-mm-dd HH:MM:SS.#####TZ

14:39:53.662522-05 is almost 2:40 p.m. CDT (e.g., in Chicago; in UTC it'd be 7:40 p.m.). The letters in the above example represent:

#### In the date part:

- YYYY – the 4-digit year.
  - mm – the zero-padded month (01–January through 12–December).
  - dd – the zero-padded day.
- In the time part:**
- HH – the zero-padded hour in a 24-hour clock.
  - MM – the minutes.
  - SS – the seconds. *Optional*.
  - ##### – the smaller parts of a second – they can be expressed using 1 to 6 digits. *Optional*.
  - ±TZ – the timezone. It must start with either + or -, and use two digits relative to UTC. *Optional*.

#### What time is it?

To answer that question in SQL, you can use:

- `CURRENT_TIME` – to find what time it is.
- `CURRENT_DATE` – to get today's date. (`GETDATE()` in SQL Server.)
- `CURRENT_TIMESTAMP` – to get the timestamp with the two above.

#### Creating values

To create a date, time, or timestamp, simply write the value as a string and cast it to the proper type.

```
SELECT CAST('2021-12-31' AS date);
SELECT CAST('15:31' AS time);
SELECT CAST('2021-12-31 23:59:29+02' AS timestamp);
```

```
SELECT CAST('15:31.124769' AS time);
```

Be careful with the last example – it will be interpreted as 15 minutes 31 seconds and 124769 microseconds! It is always a good idea to write 00 explicitly for hours: '00:15:31.124769'.

You might skip casting in simple conditions – the database will know what you mean.

```
SELECT airline, flight_number, departure_time
FROM airport_schedule
WHERE departure_time < '12:00';
```

### INTERVALS

Note: In SQL Server, intervals aren't implemented – use the `DATEADD()` and `DATEDIFF()` functions.

To get the simplest interval, subtract one time value from another:

```
SELECT CAST('2021-12-31 23:59:59' AS timestamp) - CAST('2021-06-01 12:00:00' AS timestamp);
-- result: 213 days 11:59:59
```

#### To define an interval: `INTERVAL '1' DAY`

This syntax consists of three elements: the `INTERVAL` keyword, a quoted value, and a time part keyword (singular form). You can use the following time parts: YEAR, MONTH, WEEK, DAY, HOUR, MINUTE, and SECOND. In MySQL, omit the quotes. You can join many different INTERVALs using the + or – operator:

```
INTERVAL '1' YEAR + INTERVAL '3' MONTH
```

In some databases, there's an easier way to get the above value. And it accepts plural forms! `INTERVAL '1 year 3 months'`

There are two more syntaxes in the Standard SQL:

| Syntax                       | What it does               |
|------------------------------|----------------------------|
| INTERVAL 'x-y' YEAR TO MONTH | INTERVAL 'x' year y month' |
| INTERVAL 'x-y' DAY TO SECOND | INTERVAL 'x' day y second' |

In MySQL, write `year_month` instead of `YEAR TO MONTH` and `day_second` instead of `DAY TO SECOND`.

To get the last day of a month, add one month and subtract one day:

```
SELECT CAST('2021-02-01' AS date)
+ INTERVAL '1' MONTH
- INTERVAL '1' DAY;
```

To get all events for next three months from today:

```
SELECT event_date, event_name
FROM calendar
WHERE event_date BETWEEN CURRENT_DATE AND
CURRENT_DATE + INTERVAL '3' MONTH;
```

#### To get part of the date:

```
SELECT EXTRACT(YEAR FROM birthday)
FROM artists;
```

One of possible returned values: 1946. In SQL Server, use the `DATEPART(part, date)` function.

### TIME ZONES

In the SQL Standard, the `date` type can't have an associated time zone, but the `time` and `timestamp` types can. In the real world, time zones have little meaning without the date, as the offset can vary through the year because of `daylight saving time`. So, it's best to work with the `timestamp` values.

When working with the type `timestamp` with time zone (abbr. `timestamptz`), you can type in the value in your local time zone, and it'll get converted to the UTC time zone as it is inserted into the table. Later when you select from the table it gets converted back to your local time zone. This is immune to time zone changes.

#### AT TIME ZONE

To operate between different time zones, use the `AT TIME ZONE` keyword.

If you use this format: {`timestamp` without time zone} `AT TIME ZONE` {`time zone`}, then the database will read the time stamp in the specified time zone and convert it to the time zone local to the display. It returns the time in the format `timestamp with time zone`.

If you use this format: {`timestamp` with time zone} `AT TIME ZONE` {`time zone`}, then the database will convert the time in one time zone to the target time zone specified by `AT TIME ZONE`. It returns the time in the format `timestamptz` without time zone, in the target time zone.

You can define the time zone with popular shortcuts like UTC, MST, or GMT, or by continent/city such as: America/New\_York, Europe/London, and Asia/Tokyo.

#### Examples

We set the local time zone to 'America/New\_York'.

```
SELECT TIMESTAMP '2021-07-16 21:00:00' AT
TIME ZONE 'America/Los_Angeles';
-- result: 2021-07-17 00:00:00-04
```

Here, the database takes a timestamp without a time zone and it's told it's in Los Angeles time, which is then converted to the local time – New York for displaying. This answers the question "At what time should I turn on the TV if the show starts at 9 PM in Los Angeles?"

```
SELECT TIMESTAMP WITH TIME ZONE '2021-06-20
19:30:00' AT TIME ZONE 'Australia/Sydney';
-- result: 2021-06-21 09:30:00
```

Here, the database gets a timestamp specified in the local time zone and converts it to the time in Sydney (note that it didn't return a time zone). This answers the question "What time is it in Sydney if it's 7:30 PM here?"

# Functions

- There are dozens if not hundreds of standard functions in SQL.
- All DBMS implementations have product specific functions.
- General rule:
  - If you have to do something, ask yourself
  - Am I the first one who ever had to do this?
  - If the answer is “No,” then ask Dr. Google/ChatGPT.
  - If the answer is Yes,” ask yourself, “Am I sure this is a good idea.”
- The functions are useful and straightforward.
- Some examples → To the notebook we go, yo ho!

# *Integrity Constraints*



# Integrity Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
  - A checking account must have a balance greater than \$10,000.00
  - A salary of a bank employee must be at least \$4.00 an hour
  - A customer must have a (non-null) phone number

## DFF

- Without integrity constraints in the database, maintaining data correctness requires:
  - Lots of users know what to do and do not make mistakes.
  - Dozens of programs correctly implement constraints in the code and stay up to date on changes.
- Implementing the constraints as part of the schema eliminates many issues.



# Constraints on a Single Relation

- **not null**
- **primary key**
- **unique**
- **check (P)**, where P is a predicate



# Not Null Constraints

- **not null**
  - Declare *name* and *budget* to be **not null**  
*name varchar(20) not null*  
*budget numeric(12,2) not null*



# Unique Constraints

- **unique (  $A_1, A_2, \dots, A_m$  )**
  - The unique specification states that the attributes  $A_1, A_2, \dots, A_m$  form a candidate key.
  - Candidate keys are permitted to be null (in contrast to primary keys).

# Simple Example

- Consider a simple example of an entity class *major*:
  - *major(id, name, track)*
  - “id” is a uniquely generated ID
  - “name” is the major name, e.g. “Computer Science,” “Economics,” ... ...
  - “track” is a sub-track/specialty within the major, e.g. “Applications,” “AI/ML,” ...
    - “track” is optional
    - The combination of *(name, track)* is *unique*.
- **Note:** In many DBMS, this automatically creates indices for keys/constraints.
- Switch to Notebook.

```
create table if not exists majors
(
 id int auto_increment
 primary key,
 major_name varchar(64) not null,
 major_track varchar(64) null,
 constraint table_name_pk
 unique (major_name, major_track)
);
```



# The check clause

- The **check** (P) clause specifies a predicate P that must be satisfied by every tuple in a relation.
- Example: ensure that semester is one of fall, winter, spring or summer

```
create table section
 (course_id varchar (8),
 sec_id varchar (8),
 semester varchar (6),
 year numeric (4,0),
 building varchar (15),
 room_number varchar (7),
 time_slot_id varchar (4),
 primary key (course_id, sec_id, semester, year),
 check (semester in ('Fall', 'Winter', 'Spring', 'Summer')))
```

DFF:

- We could handle the *semester check* with an *enum*.
- Switch to notebook for a slightly different example.



# Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
  - Example: If “Biology” is a department name appearing in one of the tuples in the *instructor* relation, then there exists a tuple in the *department* relation for “Biology”.
- Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the primary key of S. A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S.



## Referential Integrity (Cont.)

- Foreign keys can be specified as part of the SQL **create table** statement  
**foreign key (*dept\_name*) references *department***
- By default, a foreign key references the primary-key attributes of the referenced table.
- SQL allows a list of attributes of the referenced relation to be specified explicitly.  
**foreign key (*dept\_name*) references *department* (*dept\_name*)**



# Cascading Actions in Referential Integrity

- When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.
- An alternative, in case of delete or update is to cascade

```
create table course (
 ...
 dept_name varchar(20),
 foreign key (dept_name) references department
 on delete cascade
 on update cascade,
 ...
)
```

- Instead of cascade we can use :
  - **set null**,
  - **set default**

## DFF:

- I do not like using *cascade*. I think making changes should be explicit.
- Other people disagree.
- You will get some simple practice on HW or exams.

# *Indexes*

## *Review, Concepts, Examples*



# Index Creation

- Many queries reference only a small proportion of the records in a table.
- It is inefficient for the system to read every record to find a record with particular value
- An **index** on an attribute of a relation is a data structure that allows the database system to find those tuples in the relation that have a specified value for that attribute efficiently, without scanning through all the tuples of the relation.
- We create an index with the **create index** command

```
create index <name> on <relation-name> (attribute);
```



# Index Creation Example

- **create table student**  
*(ID varchar (5),  
name varchar (20) not null,  
dept\_name varchar (20),  
tot\_cred numeric (3,0) default 0,  
primary key (ID))*
- **create index studentID\_index on student(ID)**
- The query:

```
select *
from student
where ID = '12345'
```

can be executed by using the index to find the required record, without looking at all records of *student*

DFF:

- An index on (column1, column2, column3)
- Is also an index on (column1) and (column1, column2)
- But not (column2), (column3), (column2, column3).
- We will see “why” later in the semester.

# *Functions, Procedures, Triggers*

# *Some Concepts*



# Functions and Procedures

- Functions and procedures allow “business logic” to be stored in the database and executed from SQL statements.
- These can be defined either by the procedural component of SQL or by an external programming language such as Java, C, or C++.
- The syntax we present here is defined by the SQL standard.
  - Most databases implement nonstandard versions of this syntax.

## Note:

- The programming language, runtime and tools for functions, procedures and triggers are not easy to use.
- My view is that calling external functions is an anti-pattern (bad idea).
  - External code degrades the reliability, security and performance of the database.
  - Databases are often mission critical and the heart of environments.



# Language Constructs for Procedures & Functions

- SQL supports constructs that gives it almost all the power of a general-purpose programming language.
  - Warning: most database systems implement their own variant of the standard syntax below.
- Compound statement: **begin ... end**,
  - May contain multiple SQL statements between **begin** and **end**.
  - Local variables can be declared within a compound statements
- While and repeat statements:
  - **while** boolean expression **do**  
sequence of statements ;  
**end while**
  - **repeat**  
sequence of statements ;  
until boolean expression  
**end repeat**



## (Core) Language Constructs (Cont.)

- **For** loop
  - Permits iteration over all results of a query
- Example: Find the budget of all departments

```
declare n integer default 0;
for r as
 select budget from department
 where dept_name = 'Music'
do
 set n = n + r.budget
end for
```

### Note:

- There are various other looping constructs.



## (Core) Language Constructs – if-then-else

- Conditional statements (**if-then-else**)

```
if boolean expression
 then statement or compound statement
 elseif boolean expression
 then statement or compound statement
 else statement or compound statement
end if
```

### Note:

- We will not spend a lot of time writing functions, procedures, or triggers.
- The language and development environment are not easy to use.

# *Functions*



# Declaring SQL Functions

- Define a function that, given the name of a department, returns the count of the number of instructors in that department.

```
create function dept_count (dept_name varchar(20))
 returns integer
begin
 declare d_count integer;
 select count (*) into d_count
 from instructor
 where instructor.dept_name = dept_name
 return d_count;
end
```

- The function *dept\_count* can be used to find the department names and budget of all departments with more than 12 instructors.

```
select dept_name, budget
from department
where dept_count (dept_name) > 12
```



# Table Functions

- The SQL standard supports functions that can return tables as results; such functions are called **table functions**
- Example: Return all instructors in a given department

```
create function instructor_of (dept_name char(20))
 returns table (
 ID varchar(5),
 name varchar(20),
 dept_name varchar(20),
 salary numeric(8,2))

return table
(select ID, name, dept_name, salary
from instructor
where instructor.dept_name = instructor_of.dept_name)
```

- Usage

```
select *
from table (instructor_of ('Music'))
```

# *Procedures*



# SQL Procedures

- The *dept\_count* function could instead be written as procedure:

```
create procedure dept_count_proc (in dept_name varchar(20),
 out d_count integer)
begin
 select count(*) into d_count
 from instructor
 where instructor.dept_name = dept_count_proc.dept_name
end
```

- The keywords **in** and **out** are parameters that are expected to have values assigned to them and parameters whose values are set in the procedure in order to return results.
- Procedures can be invoked either from an SQL procedure or from embedded SQL, using the **call** statement.

```
declare d_count integer;
call dept_count_proc ('Physics', d_count);
```



## SQL Procedures (Cont.)

- Procedures and functions can be invoked also from dynamic SQL
- SQL allows more than one procedure of the same name so long as the number of arguments of the procedures with the same name is different.
- The name, along with the number of arguments, is used to identify the procedure.

# *Triggers*



# Triggers

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.
- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed.
  - Specify the actions to be taken when the trigger executes.
- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.
  - Syntax illustrated here may not work exactly on your database system; check the system manuals



# Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes
  - For example, **after update of takes on grade**
- Values of attributes before and after an update can be referenced
  - **referencing old row as** : for deletes and updates
  - **referencing new row as** : for inserts and updates
- Triggers can be activated before an event, which can serve as extra constraints. For example, convert blank grades to null.

```
create trigger setnull_trigger before update of takes
referencing new row as nrow
for each row
 when (nrow.grade = ' ')
begin atomic
 set nrow.grade = null;
end;
```



# Trigger to Maintain credits\_earned value

- **create trigger** *credits\_earned* **after update of** *takes* **on** (*grade*)  
**referencing new row as** *nrow*  
**referencing old row as** *orow*  
**for each row**  
**when** *nrow.grade*  $\neq$  'F' **and** *nrow.grade* **is not null**  
**and** (*orow.grade* = 'F' **or** *orow.grade* **is null**)  
**begin atomic**  
    **update** *student*  
    **set** *tot\_cred*= *tot\_cred* +  
        (**select** *credits*  
         **from** *course*  
         **where** *course.course\_id*= *nrow.course\_id*)  
    **where** *student.id* = *nrow.id*;  
**end;**



# Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
  - Use **for each statement** instead of **for each row**
  - Use **referencing old table** or **referencing new table** to refer to temporary tables (called ***transition tables***) containing the affected rows
  - Can be more efficient when dealing with SQL statements that update a large number of rows



# When Not To Use Triggers

- Triggers were used earlier for tasks such as
  - Maintaining summary data (e.g., total salary of each department)
  - Replicating databases by recording changes to special relations (called **change** or **delta** relations) and having a separate process that applies the changes over to a replica
- There are better ways of doing these now:
  - Databases today provide built in materialized view facilities to maintain summary data
  - Databases provide built-in support for replication
- Encapsulation facilities can be used instead of triggers in many cases
  - Define methods to update fields
  - Carry out actions as part of the update methods instead of through a trigger



# When Not To Use Triggers (Cont.)

- Risk of unintended execution of triggers, for example, when
  - Loading data from a backup copy
  - Replicating updates at a remote site
  - Trigger execution can be disabled before such actions.
- Other risks with triggers:
  - Error leading to failure of critical transactions that set off the trigger
  - Cascading execution

# *Summary*

# Comparison

## comparing triggers, functions, and procedures

|                     | triggers | functions      | stored procedures |
|---------------------|----------|----------------|-------------------|
| change data         | yes      | no             | yes               |
| return value        | never    | always         | sometimes         |
| how they are called | reaction | in a statement | exec              |

lynda.com

# Comparison – Some Details

A *trigger* has capabilities like a procedure, except ...

- You do not call it. The DB engine calls it before or after an INSERT, UPDATE, DELETE.
- The inputs are the list of incoming new, modified rows.
- The outputs are the modified versions of the new or modified rows.

| Sr.No. | User Defined Function                                                          | Stored Procedure                                                                                                                               |
|--------|--------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | Function must return a value.                                                  | Stored Procedure may or not return values.                                                                                                     |
| 2      | Will allow only Select statements, it will not allow us to use DML statements. | Can have select statements as well as DML statements such as insert, update, delete and so on                                                  |
| 3      | It will allow only input parameters, doesn't support output parameters.        | It can have both input and output parameters.                                                                                                  |
| 4      | It will not allow us to use try-catch blocks.                                  | For exception handling we can use try catch blocks.                                                                                            |
| 5      | Transactions are not allowed within functions.                                 | Can use transactions within Stored Procedures.                                                                                                 |
| 6      | We can use only table variables, it will not allow using temporary tables.     | Can use both table variables as well as temporary table in it.                                                                                 |
| 7      | Stored Procedures can't be called from a function.                             | Stored Procedures can call functions.                                                                                                          |
| 8      | Functions can be called from a select statement.                               | Procedures can't be called from Select/Where/Having and so on statements. Execute/Exec statement can be used to call/execute Stored Procedure. |
| 9      | A UDF can be used in join clause as a result set.                              | Procedures can't be used in Join clause                                                                                                        |

*Worked Example –  
Or  
Mock Prof. Ferguson While He Codes*

# Scenario

- The data model is:
  - Students
  - Faculty
  - An abstract base type Person
- We will generate some test data using Mockaroo. We will load the data.
- We will do a three table solution.
- We will implement:
  - A function to compute UNIs.
  - Triggers to automatically assign UNIs.
  - Stored procedures to create student, faculty.
  - Disable SQL operations to prevent people from making mistakes.
- Snicker at Prof. Ferguson

# *Security*

# Security Concepts (Terms from Wikipedia)

- Definitions:
  - “A (digital) identity is information on an entity used by computer systems to represent an external agent. That agent may be a person, organization, application, or device.”
  - “Authentication is the act of proving an assertion, such as the identity of a computer system user. In contrast with identification, the act of indicating a person or thing's identity, authentication is the process of verifying that identity.”
  - “Authorization is the function of specifying access rights/privileges to resources, ... More formally, "to authorize" is to define an access policy. ... During operation, the system uses the access control rules to decide whether access requests from (authenticated) consumers shall be approved (granted) or disapproved.
  - “Within an organization, roles are created for various job functions. The permissions to perform certain operations are assigned to specific roles. Members or staff (or other system users) are assigned particular roles, and through those role assignments acquire the permissions needed to perform particular system functions.”
  - “In computing, privilege is defined as the delegation of authority to perform security-relevant functions on a computer system. A privilege allows a user to perform an action with security consequences. Examples of various privileges include the ability to create a new user, install software, or change kernel functions.”
- SQL and relational database management systems implementing security by:
  - Creating identities and authentication policies.
  - Creating roles and assigning identities to roles.
  - Granting and revoking privileges to/from roles and identities.



# Authorization

- We may assign a user several forms of authorizations on parts of the database.
  - **Read** - allows reading, but not modification of data.
  - **Insert** - allows insertion of new data, but not modification of existing data.
  - **Update** - allows modification, but not deletion of data.
  - **Delete** - allows deletion of data.
- Each of these types of authorizations is called a **privilege**. We may authorize the user all, none, or a combination of these types of privileges on specified parts of a database, such as a relation or a view.



# Authorization (Cont.)

- Forms of authorization to modify the database schema
  - **Index** - allows creation and deletion of indices.
  - **Resources** - allows creation of new relations.
  - **Alteration** - allows addition or deletion of attributes in a relation.
  - **Drop** - allows deletion of relations.



# Authorization Specification in SQL

- The **grant** statement is used to confer authorization  
**grant <privilege list> on <relation or view > to <user list>**
- <user list> is:
  - a user-id
  - **public**, which allows all valid users the privilege granted
  - A role (more on this later)
- Example:
  - **grant select on department to Amit, Satoshi**
- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).



# Privileges in SQL

- **select**: allows read access to relation, or the ability to query using the view
  - Example: grant users  $U_1$ ,  $U_2$ , and  $U_3$  **select** authorization on the *instructor* relation:  

```
grant select on instructor to U1, U2, U3
```
- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges



# Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.  
**revoke <privilege list> on <relation or view> from <user list>**
- Example:  
**revoke select on student from U<sub>1</sub>, U<sub>2</sub>, U<sub>3</sub>**
- <privilege-list> may be **all** to revoke all privileges the revoker may hold.
- If <revoker-list> includes **public**, all users lose the privilege except those granted it explicitly.
- If the same privilege was granted twice to the same user by different grantors, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being revoked are also revoked.



# Roles

- A **role** is a way to distinguish among various users as far as what these users can access/update in the database.
- To create a role we use:
  - **create a role <name>**
- Example:
  - **create role instructor**
- Once a role is created we can assign “users” to the role using:
  - **grant <role> to <users>**



# Roles Example

- **create role** instructor;
- **grant** *instructor* **to** Amit;
- Privileges can be granted to roles:
  - **grant select on** *takes* **to** *instructor*;
- Roles can be granted to users, as well as to other roles
  - **create role** teaching\_assistant
  - **grant** *teaching\_assistant* **to** *instructor*;
    - *Instructor* inherits all privileges of *teaching\_assistant*
- Chain of roles
  - **create role** dean;
  - **grant** *instructor* **to** *dean*;
  - **grant** *dean* **to** Satoshi;



# Authorization on Views

- `create view geo_instructor as  
(select *  
from instructor  
where dept_name = 'Geology');`
- `grant select on geo_instructor to geo_staff`
- Suppose that a `geo_staff` member issues
  - `select *  
from geo_instructor;`
- What if
  - `geo_staff` does not have permissions on `instructor`?
  - Creator of view did not have some permissions on `instructor`?



# Other Authorization Features

- **references** privilege to create foreign key
  - **grant reference** (*dept\_name*) **on** *department* **to** Mariano;
  - Why is this required?
- transfer of privileges
  - **grant select on** *department* **to** Amit **with grant option**;
  - **revoke select on** *department* **from** Amit, Satoshi **cascade**;
  - **revoke select on** *department* **from** Amit, Satoshi **restrict**;
  - And more!

Note:

- Like in many other cases, SQL DBMS have product specific variations.

Switch to notebook.

# *REST*

# Data Modeling Concepts and REST

Almost any data model has the same core concepts:

- Types and instances:
  - Entity Type: A definition of a type of thing with properties and relationships.
  - Entity Instance: A specific instantiation of the Entity Type
  - Entity Set Instance: An Entity Type that:
    - Has properties and relationships like any entity, but ...
    - Has at least one *special relationship* – ***contains***.
- Operations, minimally CRUD, that manipulate entity types and instances:
  - Create
  - Retrieve
  - Update
  - Delete
  - Reference/Identify/... ...

## What is REST architecture?

REST stands for REpresentational State Transfer. REST is web standards based architecture and uses HTTP Protocol. It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods. REST was first introduced by Roy Fielding in 2000.

In REST architecture, a REST Server simply provides access to resources and REST client accesses and modifies the resources. Here each resource is identified by URIs/ global IDs. REST uses various representation to represent a resource like text, JSON, XML. JSON is the most popular one.

## HTTP methods

Following four HTTP methods are commonly used in REST based architecture.

- **GET** – Provides a read only access to a resource.
- **POST** – Used to create a new resource.
- **DELETE** – Used to remove a resource.
- **PUT** – Used to update a existing resource or create a new resource.

## Introduction to RESTful web services

A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. This interoperability (e.g., between Java and Python, or Windows and Linux applications) is due to the use of open standards.

Web services based on REST Architecture are known as RESTful web services. These webservices uses HTTP methods to implement the concept of REST architecture. A RESTful web service usually defines a URI, Uniform Resource Identifier a service, provides resource representation such as JSON and set of HTTP Methods.

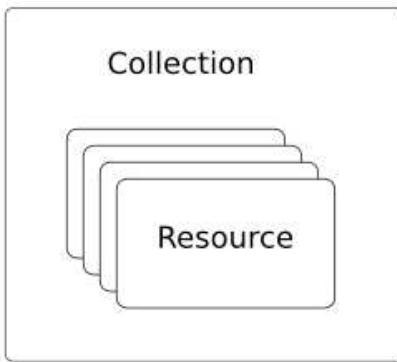
## Creating RESTful Webservice

In next chapters, we'll create a webservice say user management with following functionalities –

| Sr.No. | URI                      | HTTP Method | POST body   | Result                      |
|--------|--------------------------|-------------|-------------|-----------------------------|
| 1      | /UserService/users       | GET         | empty       | Show list of all the users. |
| 2      | /UserService/addUser     | POST        | JSON String | Add details of new user.    |
| 3      | /UserService/getUser/:id | GET         | empty       | Show details of a user.     |

# REST and Resources

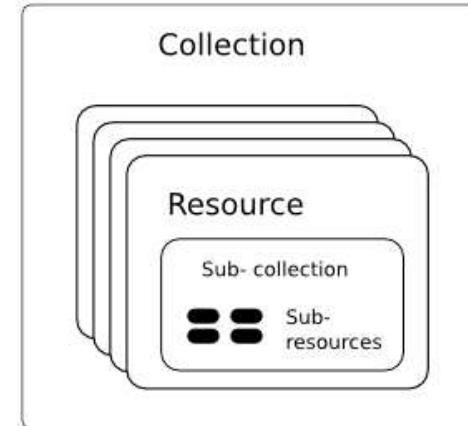
## Resource Model



A Collection with  
Resources

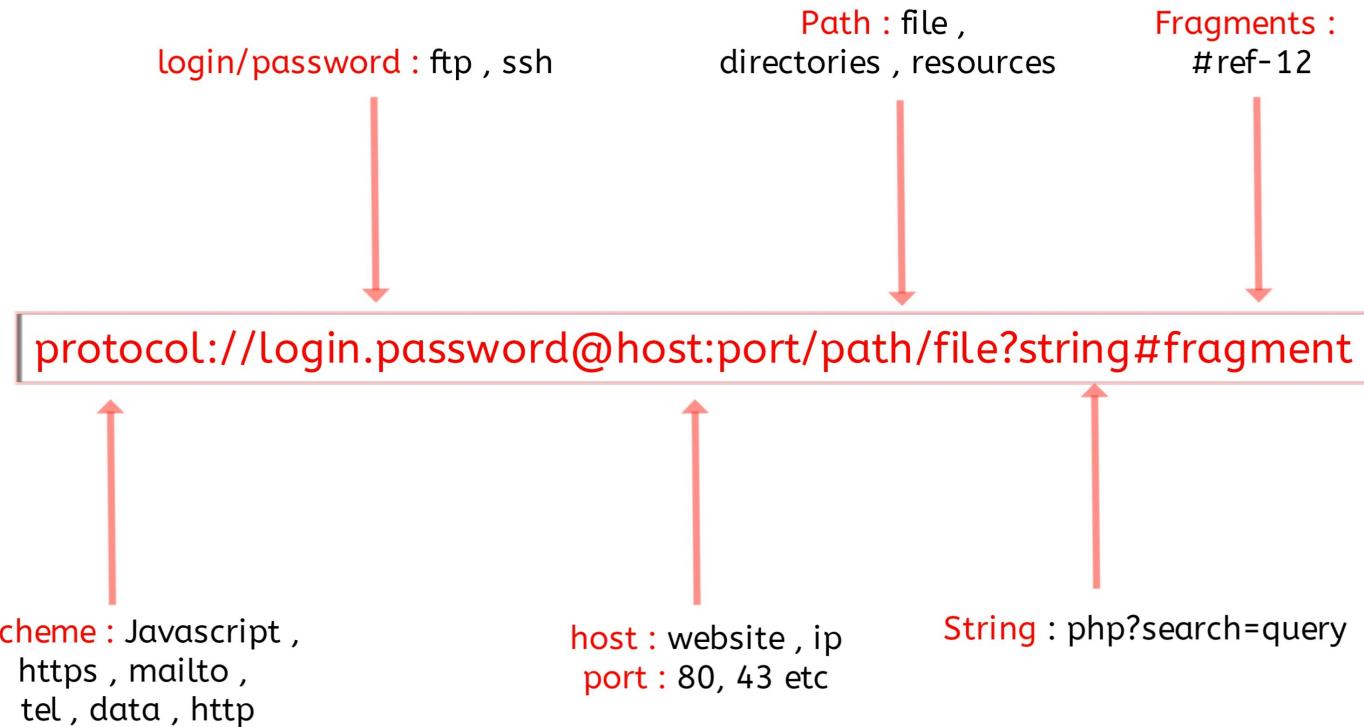


A Singleton  
Resource



Sub-collections and  
Sub-resources

# URLs



jdbc:mysql://columbia-examples.ckkqqktwkcji.us-east-1.rds.amazonaws.com:3306

# Simplistic, Conceptual Mapping (Examples)

| REST Method | Resource Path             | Relational Operation                                                                                             | DB Resource                |
|-------------|---------------------------|------------------------------------------------------------------------------------------------------------------|----------------------------|
| DELETE      | /people                   | DROP TABLE                                                                                                       | people table               |
| POST        | /people                   | INSERT INTO PEOPLE (...) VALUES(...)                                                                             | people table<br>people row |
| GET         | /people/21                | SHOW KEYS FROM people ...;<br><br>SELECT * FROM people WHERE<br>playerID= 21                                     | people row                 |
| GET         | /people/21/batting        | SELECT batting.* FROM<br>people JOIN batting USING(playerID)<br>WHERE playerID=21                                |                            |
| GET         | /people/21/batting/2004_1 | SELECT batting.* FROM<br>people JOIN batting USING(playerID)<br>WHERE playerID=21<br>AND yearID=2004 AND stint=1 |                            |

# Application Architecture

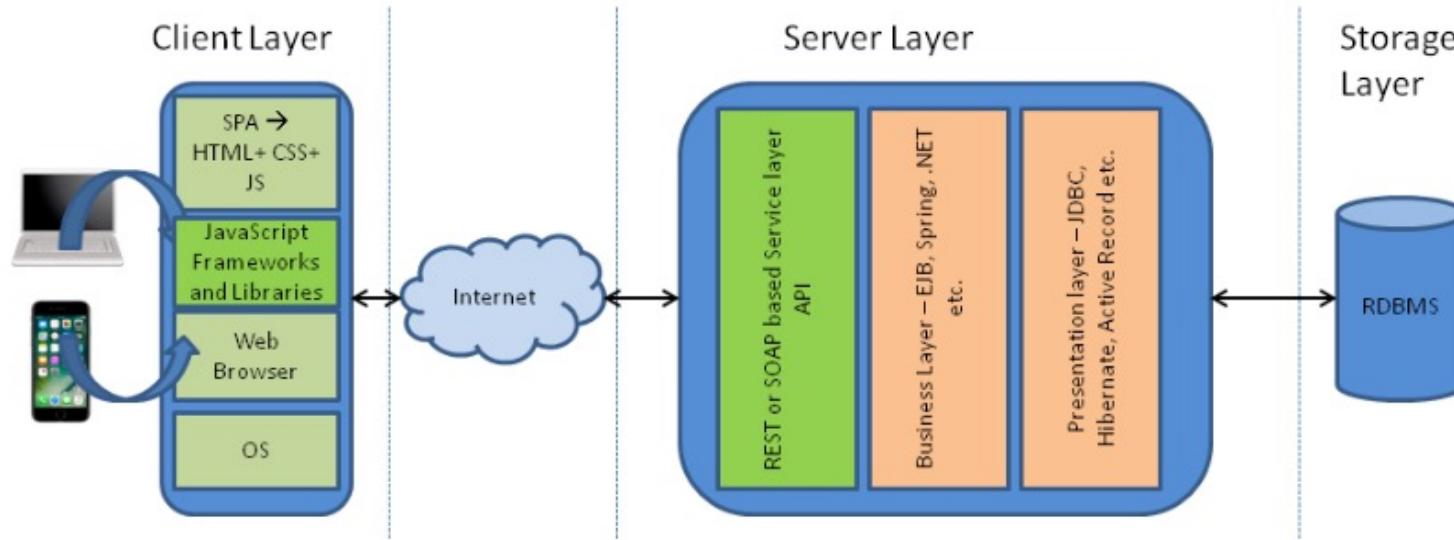


Diagram 2: The moving of the Web Layer from the Server to the Client

# *Sample Projects*

# Projects

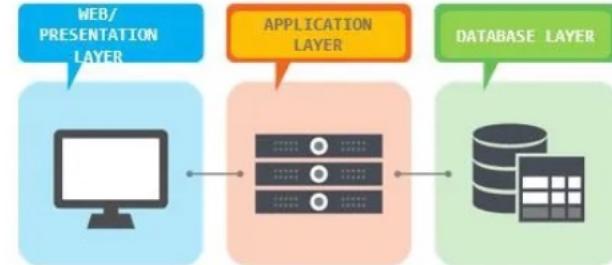
- The programming track will implement a simple, full stack web application.

## Full-stack Web Developer

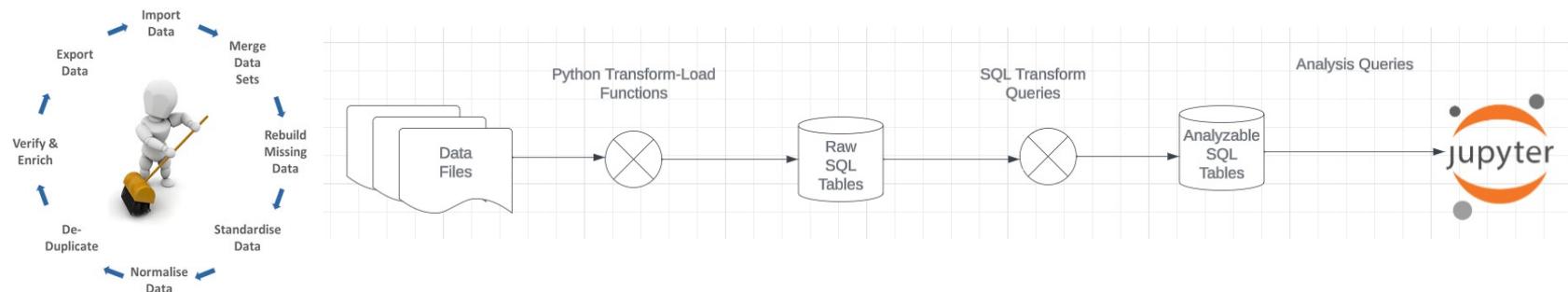
A full-stack web developer is a person who can develop both **client** and **server** software.

In addition to mastering HTML and CSS, he/she also knows how to:

- Program a **browser** (e.g. using JavaScript, jQuery, Angular, or Vue)
- Program a **server** (e.g. using PHP, ASP, Python, or Node)
- Program a **database** (e.g. using SQL, SQLite, or MongoDB)



- The non-programming track will implement a data engineering and visualization process in a Jupyter notebook.

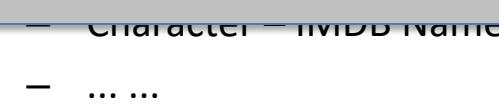


# Game of Thrones

- Bottom-Up Data Mapping:
  - “Nouns” usually map to Entity/Entity Set.
  - Nouns inside other nouns often map to:
    - Attribute
    - Relationship
  - Verbs often map to relationships.
  - Adjectives usually map to properties.
- We will start with a subset of the information:
  - Game of Thrones:
    - Episodes
    - Characters
  - IMDB:
    - names\_basics
    - title\_basics
- Entities Sets
  - Character
  - Season
  - Episode
  - Scene
  - Location, Sublocation
  - ... ...
- Relationships
  - Character – Scene
  - Character – Character (e.g. KilledBy)
  - Season – IMDB Title
  - Character – IMDB Name
  - ... ...

# Game of Thrones

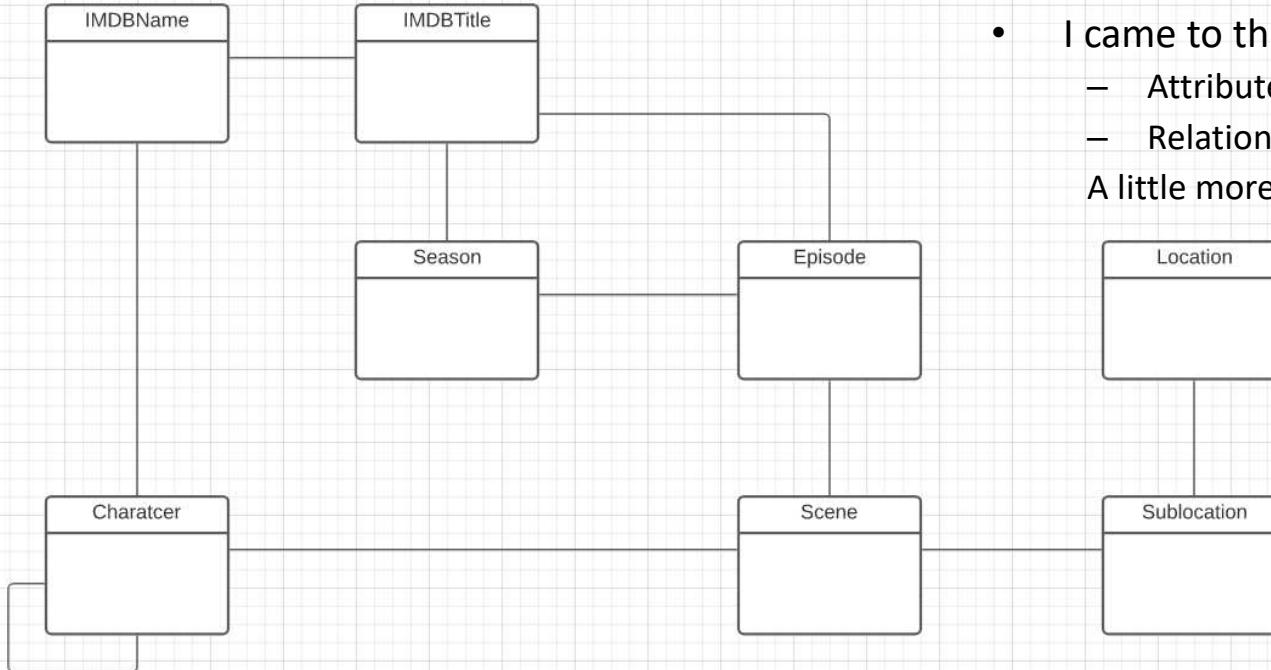
- Bottom-Up Data Mapping:
    - Entities Sets
  - We info
    - IMDB
      - 1. IMDB: <https://developer.imdb.com/non-commercial-datasets/>
        - Do not download.
        - Despite being “tiny” compared to the real world.
        - The datasets are too big for most laptops.
      - 2. Game of Thrones: <https://github.com/jeffreylancaster/game-of-thrones>
    - IMDB
      - Character – IMDB Name
      - ... ...
    - names\_basics
    - title\_basics



# Game of Thrones – Conceptual Model

Add shapes in Lucidchart ...

Add Entity Relationship Shapes



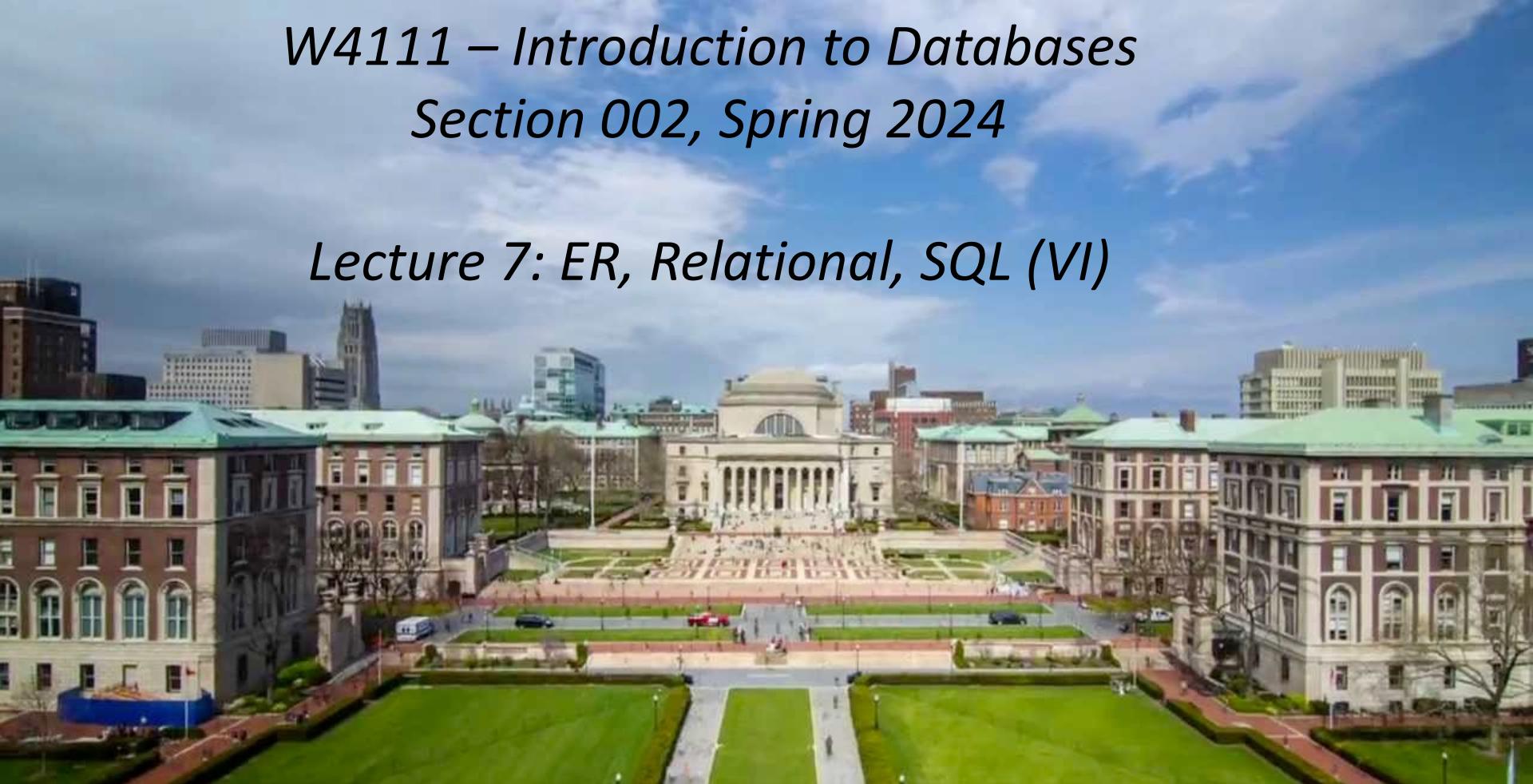
- With a little
  - Data exploration
  - Common sense
  - Judgment/experience
- I came to this conceptual model.
  - Attributes unspecified
  - Relationship required/cardinality unspecified.A little more exploration is needed.

# Sample Projects

- Walkthrough of Web Apps:
  - /Users/donaldferguson/Dropbox/000/000-A-Current-Examples/W4111-FastAPI-IMDB-Solution
  - /Users/donaldferguson/Dropbox/000/000-A-Current-Examples/current-dashboard
- Data Engineering:
  - /Users/donaldferguson/Dropbox/000/000-Columbia/2024-Spring/W4111-Intro-to-Databases-Spring-2024/Lectures/s-2024-Lecture-3/More-Data-Engineering.ipynb
  - And others.
- Sample Notebook on GoT.

*W4111 – Introduction to Databases  
Section 002, Spring 2024*

*Lecture 7: ER, Relational, SQL (VI)*



# *W4111 – Introduction to Databases*

## *Section 002, Spring 2024*

### *Lecture 7: ER, Relational, SQL (VI)*

We will start in a couple of minutes.

# *Contents*

# Contents

- WITH clause and Common Table Expressions revisited.

# *SQL*

# *WITH Clause*

## *Common Table Expressions*



# With Clause

- The **with** clause provides a way of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs.
- Find all departments with the maximum budget

```
with max_budget (value) as
 (select max(budget)
 from department)
 select department.name
 from department, max_budget
 where department.budget = max_budget.value;
```



# Complex Queries using With Clause

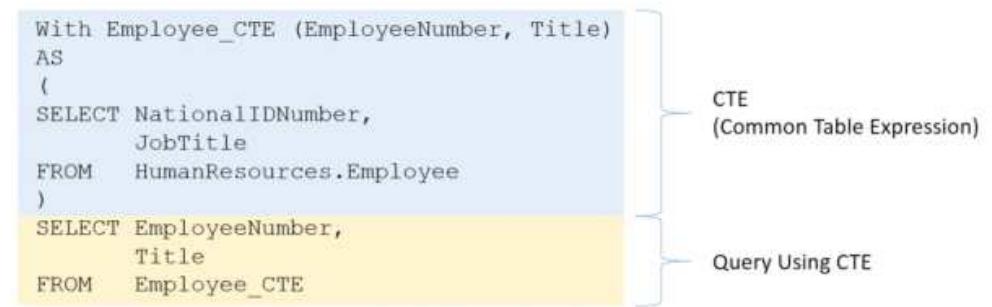
- Find all departments where the total salary is greater than the average of the total salary at all departments

```
with dept_total(dept_name, value) as
 (select dept_name, sum(salary)
 from instructor
 group by dept_name),
dept_total_avg(value) as
 (select avg(value)
 from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value > dept_total_avg.value;
```

# Common Table Expressions

- CTE: (<https://www.essentialsql.com/introduction-common-table-expressions-ctes/>)
  - “The Common Table Expressions or CTE’s for short are used within SQL (...) to simplify complex joins and subqueries, ...”
  - “A CTE (Common Table Expression) defines a temporary result set which you can then use in a SELECT statement. It becomes a convenient way to manage complicated queries.”
  - There are two types of CTEs:
    - Non-recursive.
    - Recursive (note, recursive SQL weirds me out).
  - The benefits are clarity and improved quality through incremental development.

- Basic syntax by example.
  - There may be several CTEs.
  - A CTE can reference other CTEs.



# Common Table Expressions – Example

```
WITH career_batting (playerid, h, ab, bb, hr, rbi) AS
```

```
(
 SELECT playerid, sum(h) AS h, sum(ab) AS ab, sum(bb) AS bb,
 sum(hr) AS hr, sum(rbi) AS rbi
 FROM batting GROUP BY playerid
)
,
career_pitching (playerid, w, l, ipouts, er) AS
(
 SELECT playerid, sum(w) AS w, sum(l) AS l, sum(ipouts) AS ipouts, sum(er) AS er
 FROM pitching GROUP BY playerid
)
,
career_summary (playerid, h, ab, bb, hr, rbi, bavg, obp, w, l, ipouts, er, era) AS
(
 SELECT career_batting.playerid, h, ab, bb, hr, rbi,
 IF(ab<500, NULL, round(h/ab, 3)) AS bavg,
 IF(ab<500, NULL, round((h+bb)/(ab + bb), 3)) AS obp,
 w, l, ipouts, er, round((er/(ipouts/3))*9, 3) AS era
 FROM
 career_batting JOIN career_pitching USING(playerid)
)

SELECT
 playerid, nameLast, nameFirst, career_summary.*
FROM
 people JOIN career_summary USING(playerid);
```

- Producing a career summary requires several tasks:
  1. Computing batting totals
  2. Computing pitching totals
  3. Applying formulas to produce derived averages and metrics.
  4. Joining (1), (2) and (3) into a career summary.
  5. Joining with people to bring in personal information.
- Producing the table is possible with a single SELECT statement, but ... ...
  - Developing incrementally one simpler query at a time is easier.
  - The resulting query is easier to understand and maintain.

# Common Table Expressions – Example

- Producing a career summary requires several tasks:

- I find it difficult to impossible to write complex queries without using CTEs.
  - I think about incremental steps to produce the result.
  - Write and test the CTE by adding one step at a time.
  - Please, please, please:
    - Use CTEs.
    - Use DataGrip

- Developing incrementally one simpler query at a time is easier.
- The resulting query is easier to understand and maintain.

# *Recursive Queries*



# Recursion in SQL

- SQL:1999 permits recursive view definition
- Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

```
with recursive rec_prereq(course_id, prereq_id) as (
 select course_id, prereq_id
 from prereq
 union
 select rec_prereq.course_id, prereq.prereq_id,
 from rec_rereq, prereq
 where rec_prereq.prereq_id = prereq.course_id
)
select *
from rec_prereq;
```

This example view, *rec\_prereq*, is called the *transitive closure* of the *prereq* relation



# The Power of Recursion

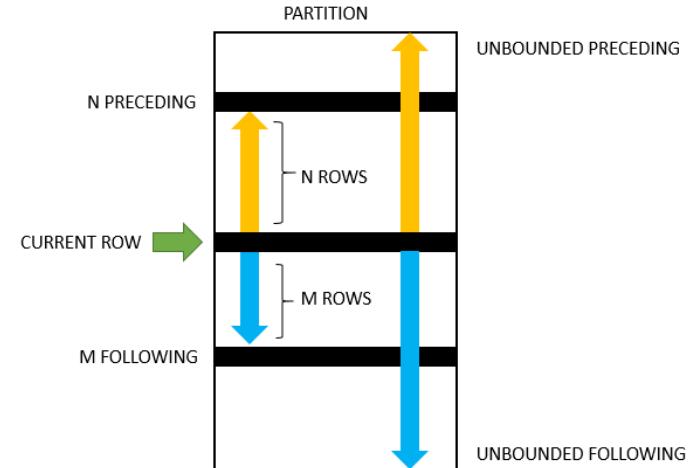
- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
  - Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of *prereq* with itself
    - This can give only a fixed number of levels of managers
    - Given a fixed non-recursive query, we can construct a database with a greater number of levels of prerequisites on which the query will not work
    - Alternative: write a procedure to iterate as many times as required
      - See procedure *findAllPrereqs* in book
- Since SQL is not Turing complete, there are many computations it cannot perform.
- Recursion enables some scenarios, but recursion in general is hard to understand.
- Switch to notebook

# *Advanced Aggregation Window Functions*

# Advanced Aggregates and Window Functions

- The aggregation functions in SQL are powerful.
- There are some scenarios that are very difficult. SQL and databases add support for more advanced capabilities:
  - Ranking
  - Windows
  - Pivot, Slide and Dice, but we will cover these with a different technology (OLAP) later in the semester.

```
<window function name>()
OVER (
 PARTITION BY <expression>
 ORDER BY <expression> [ASC | DESC]
)
```



- This is powerful, a little complex and requires trial and error.

# Switch to Notebook

- Note – This takes a lot of practice and “help.”
- To verify, in some cases you can
  - Use a subset of the data, e.g. filter by country.
  - Compute the aggregate using normal aggregation.
  - Copy into Excel/Sheets, write some formulas and verify.
- This is quite powerful. There will be a couple of take home HW/exam questions, but this takes tinkering.

# *Security*

# Security Concepts (Terms from Wikipedia)

- Definitions:
  - “A (digital) identity is information on an entity used by computer systems to represent an external agent. That agent may be a person, organization, application, or device.”
  - “Authentication is the act of proving an assertion, such as the identity of a computer system user. In contrast with identification, the act of indicating a person or thing's identity, authentication is the process of verifying that identity.”
  - “Authorization is the function of specifying access rights/privileges to resources, ... More formally, "to authorize" is to define an access policy. ... During operation, the system uses the access control rules to decide whether access requests from (authenticated) consumers shall be approved (granted) or disapproved.
  - “Within an organization, roles are created for various job functions. The permissions to perform certain operations are assigned to specific roles. Members or staff (or other system users) are assigned particular roles, and through those role assignments acquire the permissions needed to perform particular system functions.”
  - “In computing, privilege is defined as the delegation of authority to perform security-relevant functions on a computer system. A privilege allows a user to perform an action with security consequences. Examples of various privileges include the ability to create a new user, install software, or change kernel functions.”
- SQL and relational database management systems implementing security by:
  - Creating identities and authentication policies.
  - Creating roles and assigning identities to roles.
  - Granting and revoking privileges to/from roles and identities.



# Authorization

- We may assign a user several forms of authorizations on parts of the database.
  - **Read** - allows reading, but not modification of data.
  - **Insert** - allows insertion of new data, but not modification of existing data.
  - **Update** - allows modification, but not deletion of data.
  - **Delete** - allows deletion of data.
- Each of these types of authorizations is called a **privilege**. We may authorize the user all, none, or a combination of these types of privileges on specified parts of a database, such as a relation or a view.



# Authorization (Cont.)

- Forms of authorization to modify the database schema
  - **Index** - allows creation and deletion of indices.
  - **Resources** - allows creation of new relations.
  - **Alteration** - allows addition or deletion of attributes in a relation.
  - **Drop** - allows deletion of relations.



# Authorization Specification in SQL

- The **grant** statement is used to confer authorization  
**grant <privilege list> on <relation or view > to <user list>**
- <user list> is:
  - a user-id
  - **public**, which allows all valid users the privilege granted
  - A role (more on this later)
- Example:
  - **grant select on department to Amit, Satoshi**
- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).



# Privileges in SQL

- **select**: allows read access to relation, or the ability to query using the view
  - Example: grant users  $U_1$ ,  $U_2$ , and  $U_3$  **select** authorization on the *instructor* relation:  

```
grant select on instructor to U1, U2, U3
```
- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges



# Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.  
**revoke <privilege list> on <relation or view> from <user list>**
- Example:  
**revoke select on student from U<sub>1</sub>, U<sub>2</sub>, U<sub>3</sub>**
- <privilege-list> may be **all** to revoke all privileges the revoker may hold.
- If <revoker-list> includes **public**, all users lose the privilege except those granted it explicitly.
- If the same privilege was granted twice to the same user by different grantors, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being revoked are also revoked.



# Roles

- A **role** is a way to distinguish among various users as far as what these users can access/update in the database.
- To create a role we use:
  - **create a role <name>**
- Example:
  - **create role instructor**
- Once a role is created we can assign “users” to the role using:
  - **grant <role> to <users>**



# Roles Example

- **create role** instructor;
- **grant** *instructor* **to** Amit;
- Privileges can be granted to roles:
  - **grant select on** *takes* **to** *instructor*;
- Roles can be granted to users, as well as to other roles
  - **create role** teaching\_assistant
  - **grant** *teaching\_assistant* **to** *instructor*;
    - *Instructor* inherits all privileges of *teaching\_assistant*
- Chain of roles
  - **create role** dean;
  - **grant** *instructor* **to** *dean*;
  - **grant** *dean* **to** Satoshi;



# Authorization on Views

- `create view geo_instructor as  
(select *  
from instructor  
where dept_name = 'Geology');`
- `grant select on geo_instructor to geo_staff`
- Suppose that a `geo_staff` member issues
  - `select *  
from geo_instructor;`
- What if
  - `geo_staff` does not have permissions on `instructor`?
  - Creator of view did not have some permissions on `instructor`?



# Other Authorization Features

- **references** privilege to create foreign key
  - **grant reference** (*dept\_name*) **on** *department* **to** Mariano;
  - Why is this required?
- transfer of privileges
  - **grant select on** *department* **to** Amit **with grant option**;
  - **revoke select on** *department* **from** Amit, Satoshi **cascade**;
  - **revoke select on** *department* **from** Amit, Satoshi **restrict**;
  - And more!

Note:

- Like in many other cases, SQL DBMS have product specific variations.

Switch to notebook.

*Worked Example –  
Or  
Mock Prof. Ferguson While He Codes*

# Scenario

- The data model is:
  - Students
  - Faculty
  - An abstract base type Person
- We will
  - Draw the logical and a partial physical ER diagram.
  - ~~– Generate some test data using Mockaroo. We will load the data and create the initial schema.~~
  - Give you a bunch of data. --> Load → Tables → constraints
  - Implement a three-table solution, including indexes, views and constraints.
  - A function to compute UNIs.
  - Triggers to automatically assign UNIs.
  - Stored procedures to create student, faculty.
  - Create non-root users.
  - Revoke SQL operations to prevent people from making mistakes.
- Snicker at Prof. Ferguson. I put this last, but it will like occur during all steps.

# *REST*

# Data Modeling Concepts and REST

Almost any data model has the same core concepts:

- Types and instances:
  - Entity Type: A definition of a type of thing with properties and relationships.
  - Entity Instance: A specific instantiation of the Entity Type
  - Entity Set Instance: An Entity Type that:
    - Has properties and relationships like any entity, but ...
    - Has at least one *special relationship* – ***contains***.
- Operations, minimally CRUD, that manipulate entity types and instances:
  - Create
  - Retrieve
  - Update
  - Delete
  - Reference/Identify/... ...

## What is REST architecture?

REST stands for REpresentational State Transfer. REST is web standards based architecture and uses HTTP Protocol. It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods. REST was first introduced by Roy Fielding in 2000.

In REST architecture, a REST Server simply provides access to resources and REST client accesses and modifies the resources. Here each resource is identified by URIs/ global IDs. REST uses various representation to represent a resource like text, JSON, XML. JSON is the most popular one.

## HTTP methods

Following four HTTP methods are commonly used in REST based architecture.

- **GET** – Provides a read only access to a resource.
- **POST** – Used to create a new resource.
- **DELETE** – Used to remove a resource.
- **PUT** – Used to update a existing resource or create a new resource.

## Introduction to RESTful web services

A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. This interoperability (e.g., between Java and Python, or Windows and Linux applications) is due to the use of open standards.

Web services based on REST Architecture are known as RESTful web services. These webservices uses HTTP methods to implement the concept of REST architecture. A RESTful web service usually defines a URI, Uniform Resource Identifier a service, provides resource representation such as JSON and set of HTTP Methods.

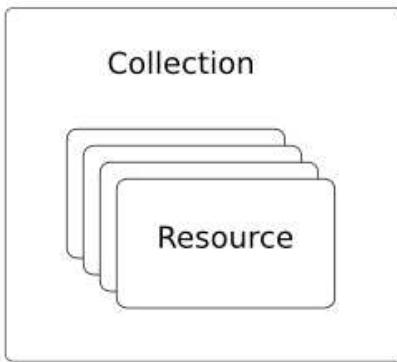
## Creating RESTful Webservice

In next chapters, we'll create a webservice say user management with following functionalities –

| Sr.No. | URI                      | HTTP Method | POST body   | Result                      |
|--------|--------------------------|-------------|-------------|-----------------------------|
| 1      | /UserService/users       | GET         | empty       | Show list of all the users. |
| 2      | /UserService/addUser     | POST        | JSON String | Add details of new user.    |
| 3      | /UserService/getUser/:id | GET         | empty       | Show details of a user.     |

# REST and Resources

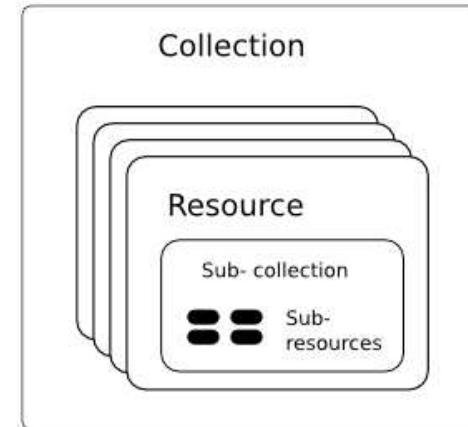
## Resource Model



A Collection with  
Resources

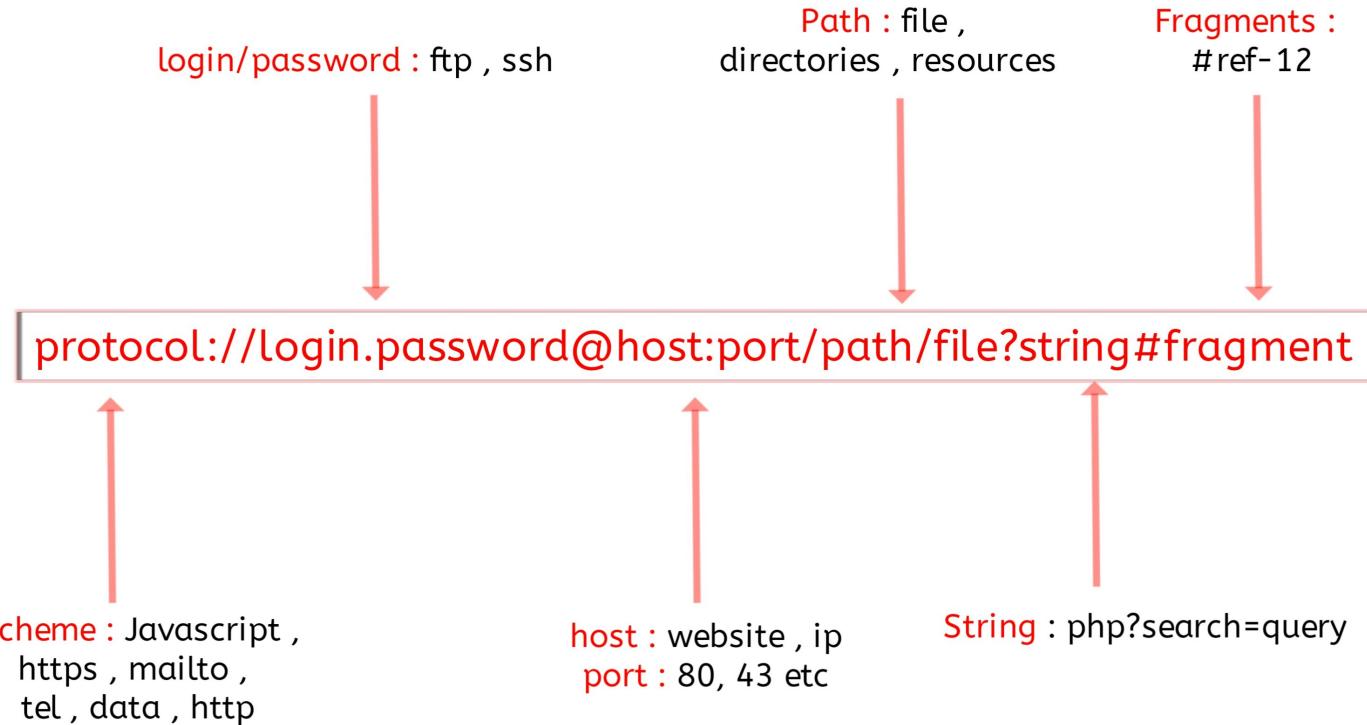


A Singleton  
Resource



Sub-collections and  
Sub-resources

# URLs



# Simplistic, Conceptual Mapping (Examples)

| REST Method | Resource Path             | Relational Operation                                                                                             | DB Resource                |
|-------------|---------------------------|------------------------------------------------------------------------------------------------------------------|----------------------------|
| DELETE      | /people                   | DROP TABLE                                                                                                       | people table               |
| POST        | /people                   | INSERT INTO PEOPLE (...) VALUES(...)                                                                             | people table<br>people row |
| GET         | /people/21                | SHOW KEYS FROM people ...;<br><br>SELECT * FROM people WHERE<br>playerID= 21                                     | people row                 |
| GET         | /people/21/batting        | SELECT batting.* FROM<br>people JOIN batting USING(playerID)<br>WHERE playerID=21                                |                            |
| GET         | /people/21/batting/2004_1 | SELECT batting.* FROM<br>people JOIN batting USING(playerID)<br>WHERE playerID=21<br>AND yearID=2004 AND stint=1 |                            |

# Application Architecture

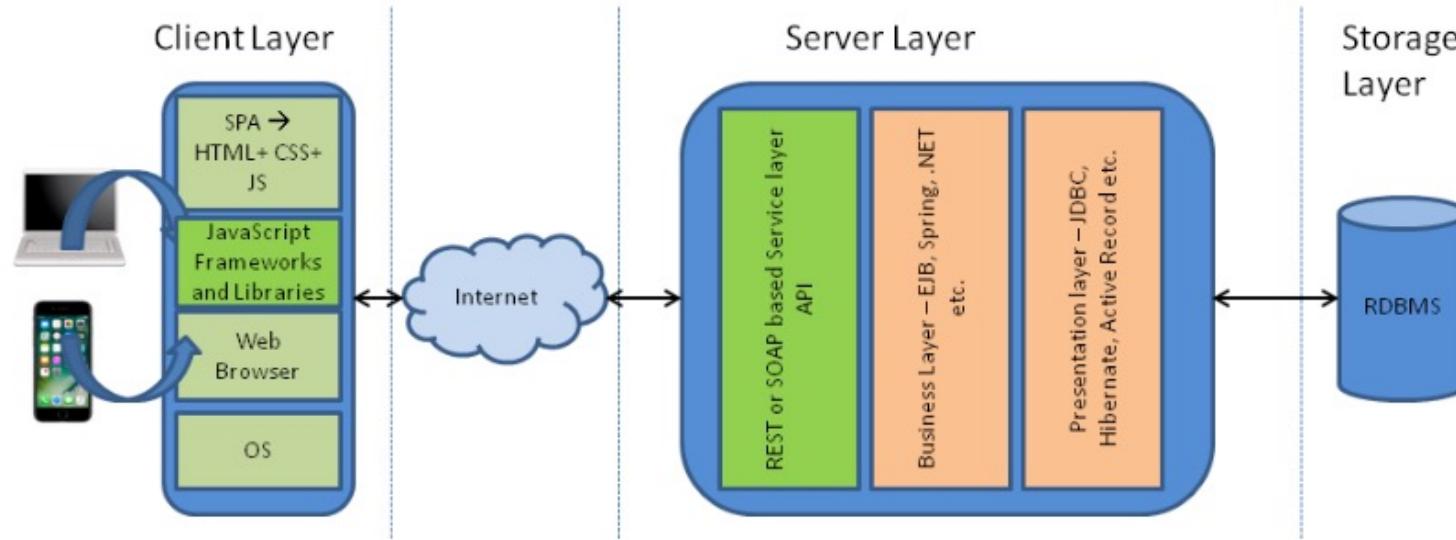


Diagram 2: The moving of the Web Layer from the Server to the Client

# Walkthrough Code

- /Users/donaldferguson/Dropbox/000/000-A-Current-Examples/W4111-FastAPI-IMDB-Solution
- /Users/donaldferguson/Dropbox/000/000-A-Current-Examples/current-dashboard

*Big Data  
Business Intelligence  
Decision Support  
(Preview to enable project)*



# Decision Support Systems

- **Decision-support systems** are used to make business decisions, often based on data collected by on-line transaction-processing systems.
- Examples of business decisions:
  - What items to stock?
  - What insurance premium to change?
  - To whom to send advertisements?
- Examples of data used for making decisions
  - Retail sales transaction details
  - Customer profiles (income, age, gender, etc.)



# Decision-Support Systems: Overview

- **Data analysis** tasks are simplified by specialized tools and SQL extensions
  - Example tasks
    - ▶ For each product category and each region, what were the total sales in the last quarter and how do they compare with the same quarter last year
    - ▶ As above, for each product category and each customer category
- **Statistical analysis** packages (e.g., : S++) can be interfaced with databases
  - Statistical analysis is a large field, but not covered here
- **Data mining** seeks to discover knowledge automatically in the form of statistical rules and patterns from large databases.
- A **data warehouse** archives information gathered from multiple sources, and stores it under a unified schema, at a single site.
  - Important for large businesses that generate data from multiple divisions, possibly at multiple sites
  - Data may also be purchased externally

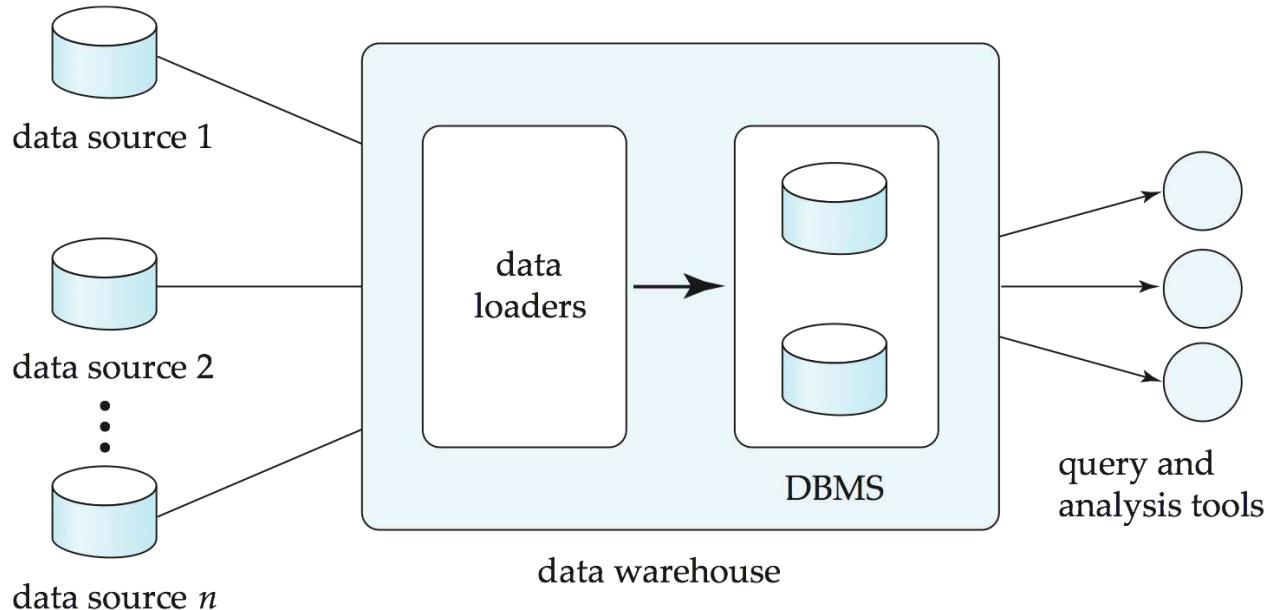


# Data Warehousing

- Data sources often store only current data, not historical data
- Corporate decision making requires a unified view of all organizational data, including historical data
- A **data warehouse** is a repository (archive) of information gathered from multiple sources, stored under a unified schema, at a single site
  - Greatly simplifies querying, permits study of historical trends
  - Shifts decision support query load away from transaction processing systems



# Data Warehousing



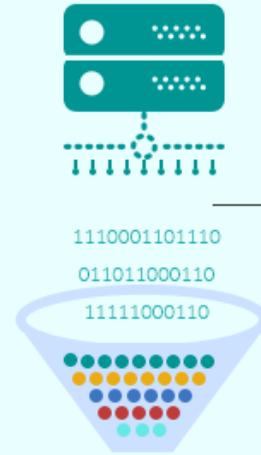
# Data Warehouse vs Data Lake

<https://www.grazitti.com/blog/data-lake-vs-data-warehouse-which-one-should-you-go-for/>

## DATA WAREHOUSE



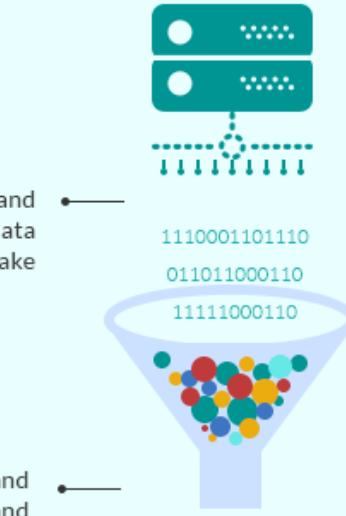
## DATA LAKE



- Data is processed and organized into a single schema before being put into the warehouse

- The analysis is done on the cleansed data in the warehouse

Raw and unstructured data goes into a data lake



Data is selected and organized as and when needed

# Simplistic: Data Warehouse vs Data Lake

<https://panoply.io/data-warehouse-guide/data-warehouse-vs-data-lake/>

| DATA WAREHOUSE                   | vs.        | DATA LAKE                                         |
|----------------------------------|------------|---------------------------------------------------|
| structured, processed            | DATA       | structured / semi-structured / unstructured, raw  |
| schema-on-write                  | PROCESSING | schema-on-read                                    |
| expensive for large data volumes | STORAGE    | designed for low-cost storage                     |
| less agile, fixed configuration  | AGILITY    | highly agile, configure and reconfigure as needed |
| mature                           | SECURITY   | maturing                                          |
| business professionals           | USERS      | data scientists et. al.                           |



# Design Issues

## ■ When and how to gather data

- **Source driven architecture**: data sources transmit new information to warehouse, either continuously or periodically (e.g., at night)
- **Destination driven architecture**: warehouse periodically requests new information from data sources
- Keeping warehouse exactly synchronized with data sources (e.g., using two-phase commit) is too expensive
  - ▶ Usually OK to have slightly out-of-date data at warehouse
  - ▶ Data/updates are periodically downloaded from online transaction processing (OLTP) systems.

## ■ What schema to use

- Schema integration



# More Warehouse Design Issues

## ■ *Data cleansing*

- E.g., correct mistakes in addresses (misspellings, zip code errors)
- **Merge** address lists from different sources and **purge** duplicates

## ■ *How to propagate updates*

- Warehouse schema may be a (materialized) view of schema from data sources

## ■ *What data to summarize*

- Raw data may be too large to store on-line
- Aggregate values (totals/subtotals) often suffice
- Queries on raw data can often be transformed by query optimizer to use aggregate values

# *Sample Projects*

# Projects

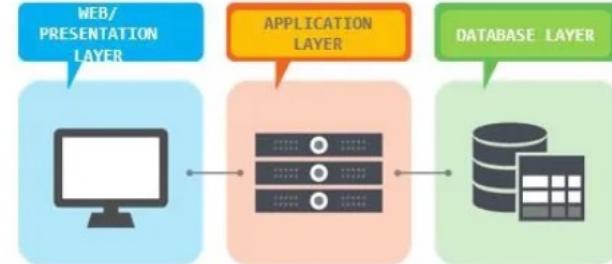
- The programming track will implement a simple, full stack web application.

## Full-stack Web Developer

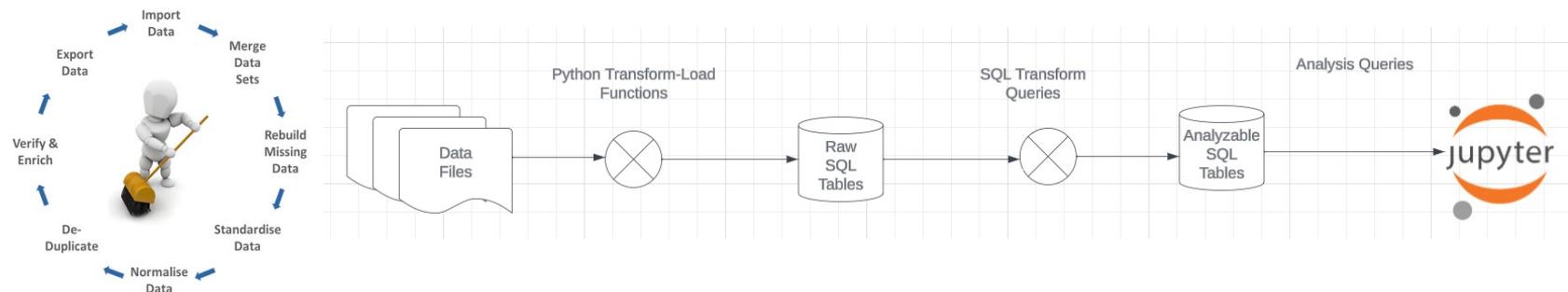
A full-stack web developer is a person who can develop both **client** and **server** software.

In addition to mastering HTML and CSS, he/she also knows how to:

- Program a **browser** (e.g. using JavaScript, jQuery, Angular, or Vue)
- Program a **server** (e.g. using PHP, ASP, Python, or Node)
- Program a **database** (e.g. using SQL, SQLite, or MongoDB)



- The non-programming track will implement a data engineering and visualization process in a Jupyter notebook via a simple data warehouse.



# Game of Thrones

- Bottom-Up Data Mapping:
  - “Nouns” usually map to Entity/Entity Set.
  - Nouns inside other nouns often map to:
    - Attribute
    - Relationship
  - Verbs often map to relationships.
  - Adjectives usually map to properties.
- We will start with a subset of the information:
  - Game of Thrones:
    - Episodes
    - Characters
  - IMDB:
    - names\_basics
    - title\_basics
- Entities Sets
  - Character
  - Season
  - Episode
  - Scene
  - Location, Sublocation
  - ... ...
- Relationships
  - Character – Scene
  - Character – Character (e.g. KilledBy)
  - Season – IMDB Title
  - Character – IMDB Name
  - ... ...

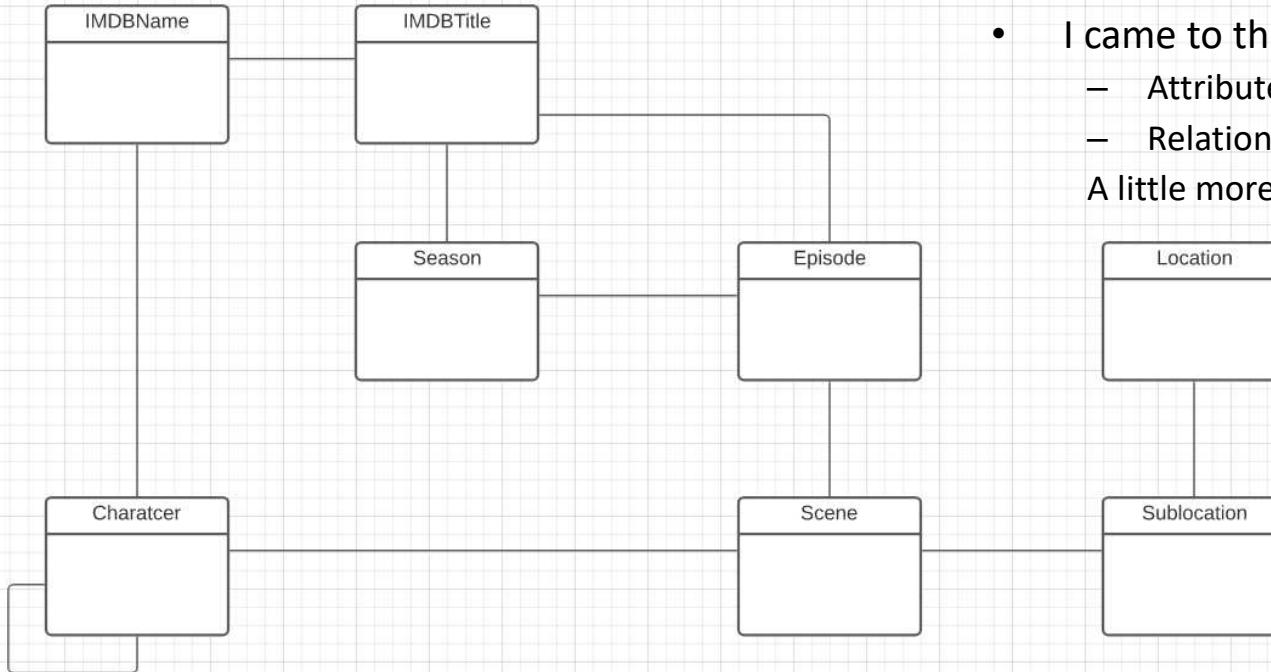
# Game of Thrones



# Game of Thrones – Conceptual Model

Add shapes in Lucidchart ...

Add Entity Relationship Shapes



- With a little
  - Data exploration
  - Common sense
  - Judgment/experience
- I came to this conceptual model.
  - Attributes unspecified
  - Relationship required/cardinality unspecified.A little more exploration is needed.

# Sample Projects

- Walkthrough of Web Apps:
  - /Users/donaldferguson/Dropbox/000/000-A-Current-Examples/W4111-FastAPI-IMDB-Solution
  - /Users/donaldferguson/Dropbox/000/000-A-Current-Examples/current-dashboard
- Data Engineering:
  - /Users/donaldferguson/Dropbox/000/000-Columbia/2024-Spring/W4111-Intro-to-Databases-Spring-2024/Lectures/s-2024-Lecture-3/More-Data-Engineering.ipynb
  - And others.
- Sample Notebook on GoT.

*W4111 – Introduction to Databases*  
*Section 002, Spring 2024*  
*Lecture 8*  
*Module II (1), NoSQL (1)*



*W4111 – Introduction to Databases*  
*Section 002, Spring 2024*  
*Lecture 8*  
*Module II (1), NoSQL (1)*

We will start in a couple of minutes.

# *Module II Kickoff*

# Course Modules – Reminder

## Course Overview

Each section of W4111 is slightly different based on student interest and professor's focus. There is a common, core syllabus. Professors cover topics in different orders and grouping based on teaching style.

This section of W4111 has four modules:

- **Foundational concepts (50% of semester):** This module covers concepts like data models, relational model, relational databases and applications, schema, normalization, ... The module focuses on the relational model and relational databases. The concepts are critical and foundational for all types of databases and data centric applications.
- **Database management system architecture and implementation (10%):** This module covers the software architecture, algorithms and implementation techniques that allow [databases management systems](#) to deliver functions. Topics include memory hierarchy, storage systems, caching/buffer pools, indexes, query processing, query optimization, transaction processing, isolation and concurrency control.
- **NoSQL – “Not Only SQL” databases (20%):** This module provides motivation for [“NoSQL”](#) data models and databases, and covers examples and use cases. The module also includes cloud databases and databases-as-a-service.
- **Data Enabled Decision Support (20%):** This module covers data warehouses, data import and cleanse, OLAP, Pivot Tables, Star Schema, reporting and visualization, and provides an overview of analysis techniques, e.g. clustering, classification, analysis, mining.

# *Module II – DBMS Architecture and Implementation Overview and Reminder*

# Module II – DBMS Architecture and

What is a database? In essence a database is nothing more than a collection of information that exists over a long period of time, often many years. In common parlance, the term *database* refers to a collection of data that is managed by a DBMS. The DBMS is expected to:

- 
1. Allow users to create new databases and specify their *schemas* (logical structure of the data), using a specialized *data-definition language*.

Covered for the relational model.

**Database Systems: The Complete Book (2nd Edition)**

by [Hector Garcia-Molina](#) (Author), [Jeffrey D. Ullman](#) (Author), [Jennifer Widom](#) (Author)

# Module II – DBMS Architecture and

- 
- 
2. Give users the ability to *query* the data (a “query” is database lingo for a question about the data) and modify the data, using an appropriate language, often called a *query language* or *data-manipulation language*.
  3. Support the storage of very large amounts of data — many terabytes or more — over a long period of time, allowing efficient access to the data for queries and database modifications.
  4. Enable *durability*, the recovery of the database in the face of failures, errors of many kinds, or intentional misuse.
  5. Control access to data from many users at once, without allowing unexpected interactions among users (called *isolation*) and without actions on the data to be performed partially but not completely (called *atomicity*).

**Database Systems: The Complete Book (2nd Edition)**

by [Hector Garcia-Molina](#) (Author), [Jeffrey D. Ullman](#) (Author), [Jennifer Widom](#) (Author)



# Purpose of Database Systems

In the early days, database applications were built directly on top of file systems, which leads to:

- Data redundancy and inconsistency: data is stored in multiple file formats resulting in 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

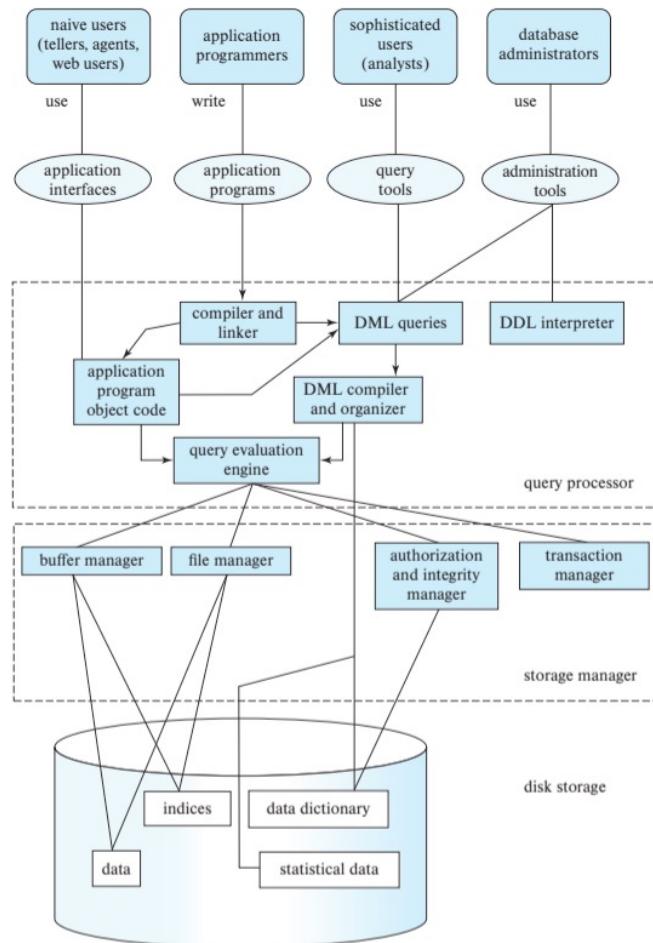
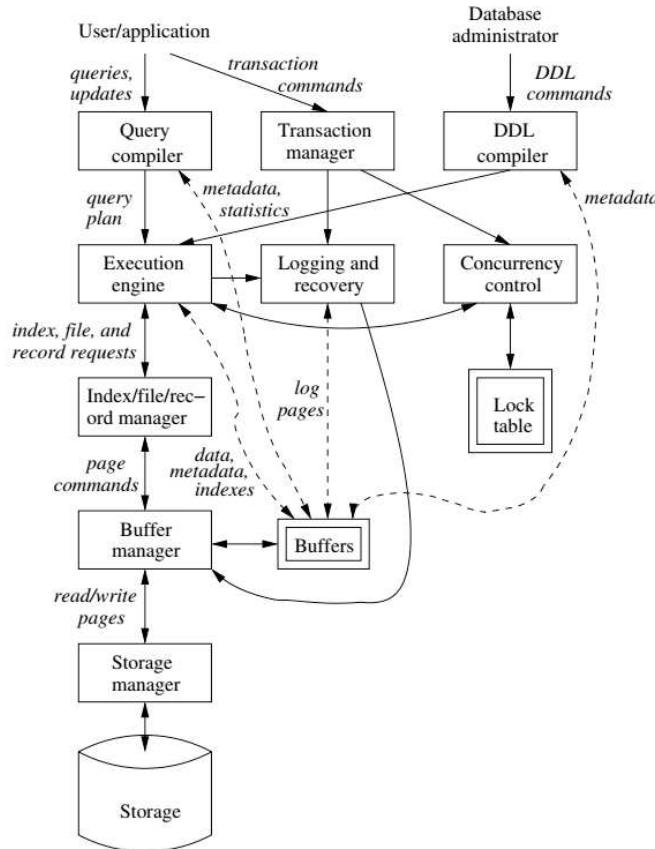


# Purpose of Database Systems (Cont.)

- 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
    - Ex: 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

**Database systems offer solutions to all the above problems**

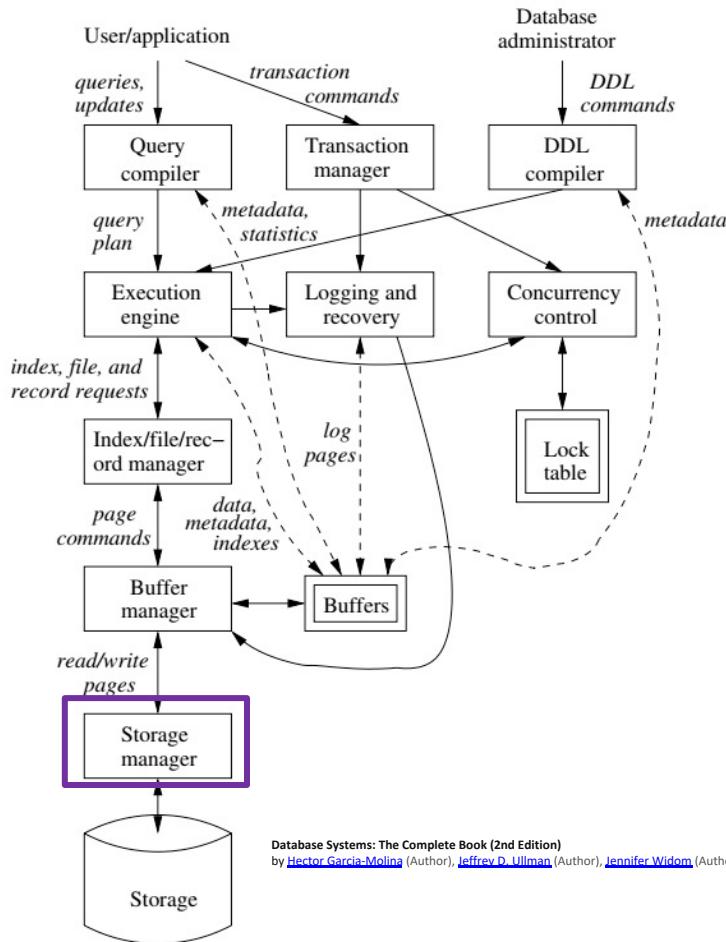
# DBMS Arch.



# Data Management

## Today

- Load/save things quickly.



# *Disks*

## *Input/Output (IO)*

# Disks as Far as the Eye can See

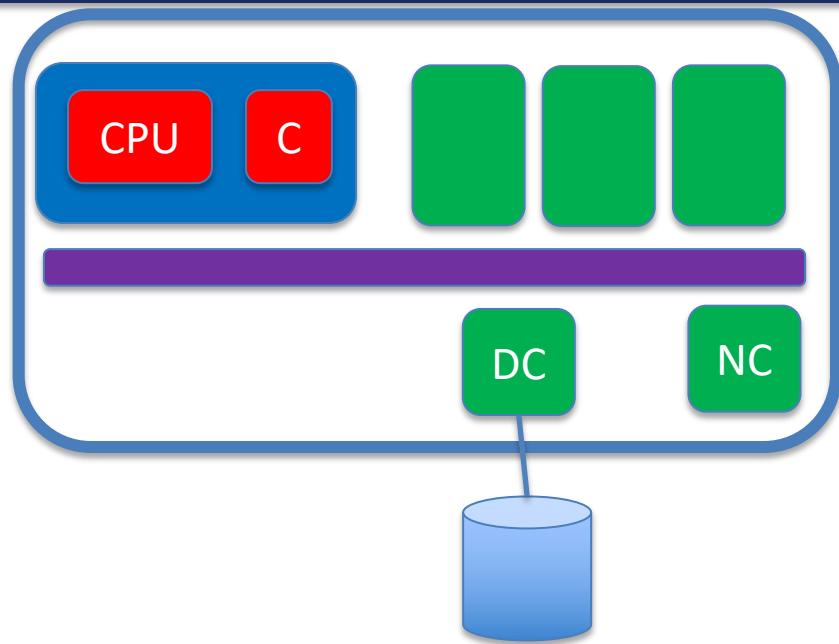
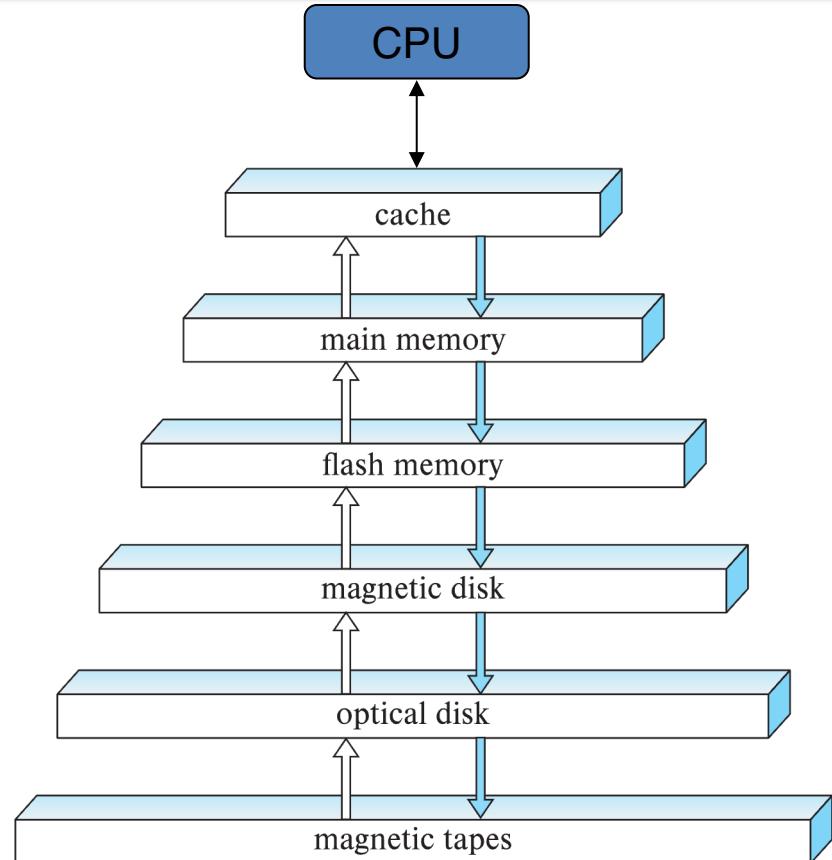




# Classification of Physical Storage Media

- Can differentiate storage into:
  - **volatile storage:** loses contents when power is switched off
  - **non-volatile storage:**
    - Contents persist even when power is switched off.
    - Includes secondary and tertiary storage, as well as battery-backed up main-memory.
- Factors affecting choice of storage media include
  - Speed with which data can be accessed
  - Cost per unit of data
  - Reliability

# Storage Hierarchy



From: Database System Concepts, 7<sup>th</sup> Ed.

# Memory Hierarchy

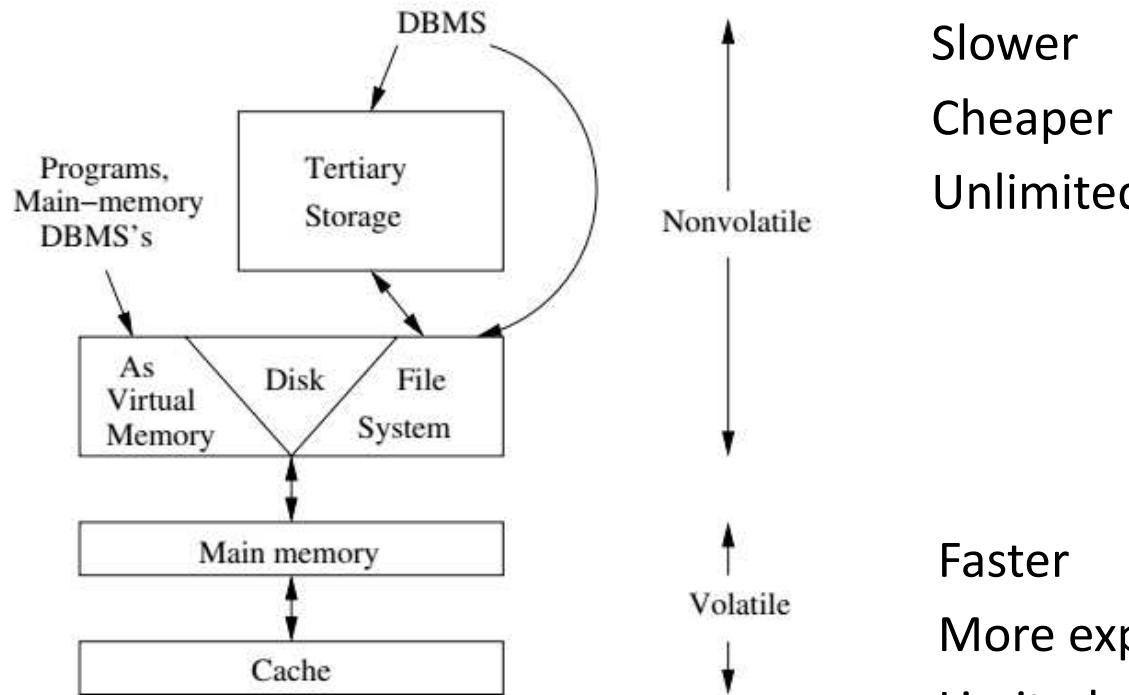


Figure 13.1: The memory hierarchy

From: Database System Concepts, 7<sup>th</sup> Ed.

# Memory Hierarchy (Very Old Numbers – Still Directionally Valid)

## Storage Technology

Price, Performance & Capacity

| Technologies     | Capacity (GB) | Latency (microS) | IOPs    | Cost/IOPS (\$) | Cost/GB (\$) |
|------------------|---------------|------------------|---------|----------------|--------------|
| Cloud Storage    | Unlimited     | 60,000           | 20      | 17c/GB         | 0.15/month   |
| Capacity HDDs    | 2,500         | 12,000           | 250     | 1.67           | 0.15         |
| Performance HDDs | 300           | 7,000            | 500     | 1.52           | 1.30         |
| SSDs (write)     | 64            | 300              | 5000    | 0.20           | 13           |
| SSDs (read only) | 64            | 45               | 30,000  | 0.03           | 13           |
| DRAM             | 8             | 0.005            | 500,000 | 0.001          | 52           |

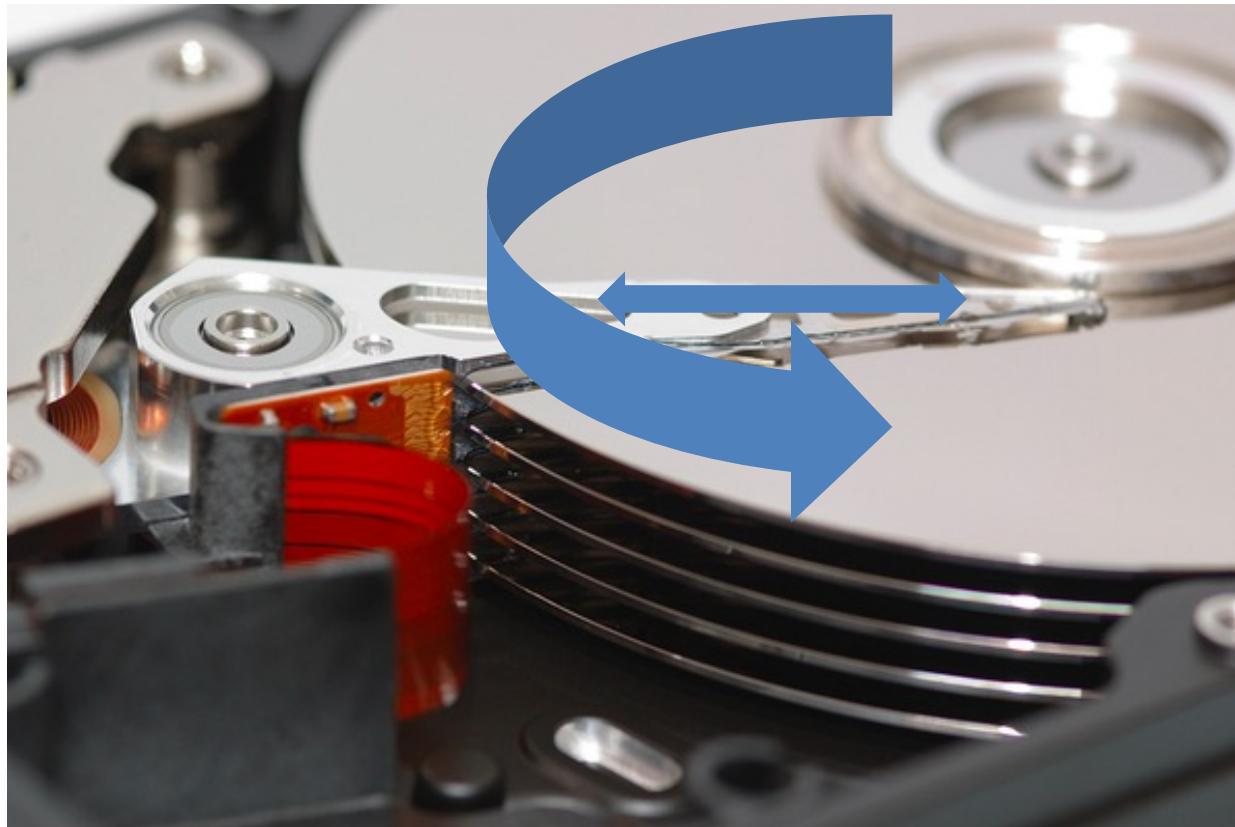
- These numbers are ancient.
- Looking for more modern numbers.
- But, does give an idea of
  - Price
  - Performance
- The general observation is that
  - Performance goes up 10X/level.
  - Price goes up 10x per level.
- Note: One major change is improved price performance of SSD relative to HDD for large data.



# Storage Hierarchy (Cont.)

- **primary storage:** Fastest media but volatile (cache, main memory).
- **secondary storage:** next level in hierarchy, non-volatile, moderately fast access time
  - Also called **on-line storage**
  - E.g., flash memory, magnetic disks
- **tertiary storage:** lowest level in hierarchy, non-volatile, slow access time
  - also called **off-line storage** and used for **archival storage**
  - e.g., magnetic tape, optical storage
  - Magnetic tape
    - Sequential access, 1 to 12 TB capacity
    - A few drives with many tapes
    - Juke boxes with petabytes (1000's of TB) of storage

# Hard Disk Drive



# Disk Configuration

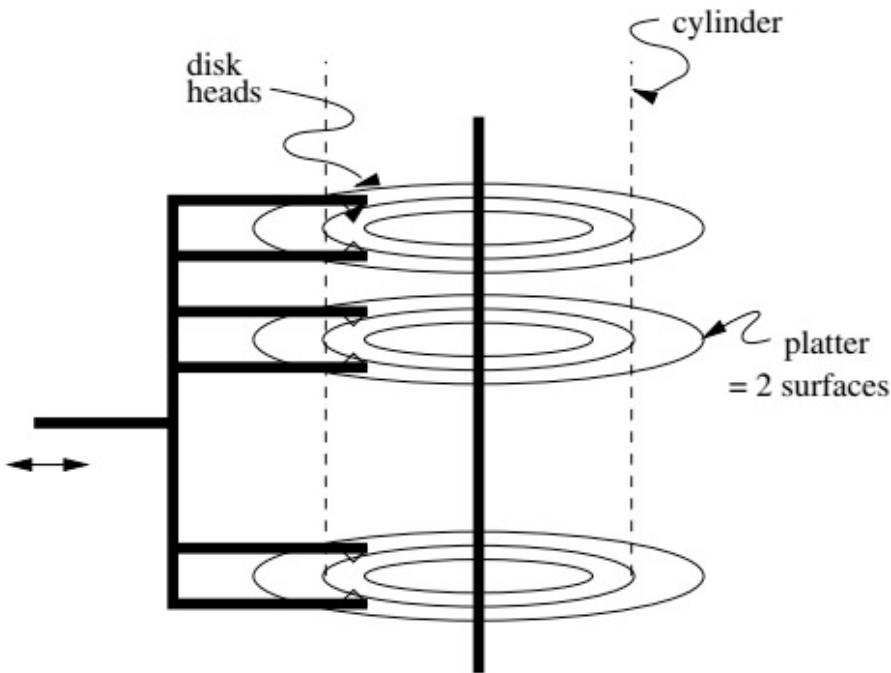


Figure 13.2: A typical disk

## Components of disk I/O delay

Seek: Move head to cylinder/track.

Rotation: Wait for sector to get under head

Transfer: Move data from disk to memory.

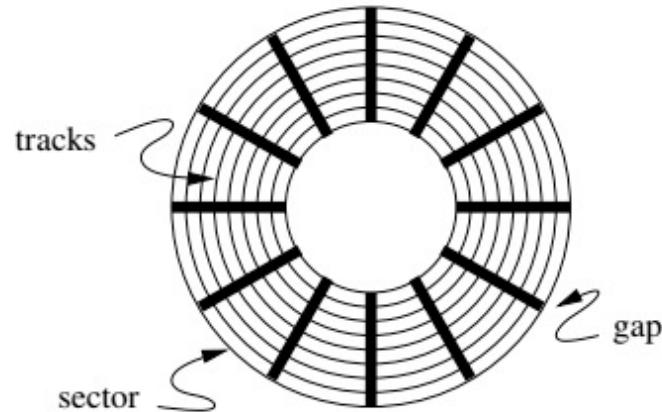
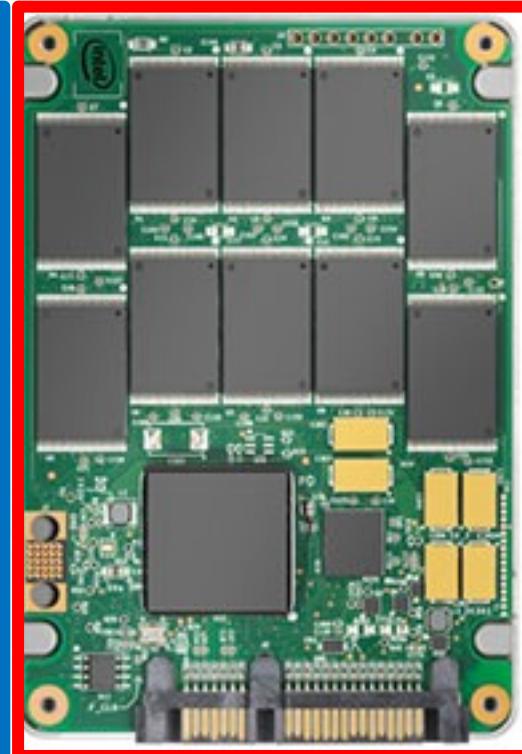


Figure 13.3: Top view of a disk surface

Database Systems: The Complete Book (2nd Edition)  
by [Hector Garcia-Molina](#) (Author), [Jeffrey D. Ullman](#) (Author), [Jennifer Widom](#) (Author)

# Hard Disk versus Solid State Disk

Hard  
Disk  
Drive



Solid  
State  
Drive



# Flash Storage

- NOR flash vs NAND flash
- NAND flash
  - used widely for storage, cheaper than NOR flash
  - requires page-at-a-time read (page: 512 bytes to 4 KB)
    - 20 to 100 microseconds for a page read
    - Not much difference between sequential and random read
  - Page can only be written once
    - Must be erased to allow rewrite
- **Solid state disks**
  - Use standard block-oriented disk interfaces, but store data on multiple flash storage devices internally
  - Transfer rate of up to 500 MB/sec using SATA, and up to 3 GB/sec using NVMe PCIe

# Logical Block Addressing

- Concept:
  - The *unit of transfer* from a “disk” to the computers memory is a “block.  
Blocks are usually relatively large, e.g. 16 KB, 32 KB, ... ...
  - A program that reads or write a single byte, requires the database engine (or file system) to read/write the entire block.
- The address of a block in the entire space of blocks is:
  - (Device ID, Block ID)
  - Block ID is simple 0, 1, 2, ... ...
- The disk controller and disk implementation translate the *logical block address* into the *physical address of blocks*.
- The physical address changes over time for various reasons, e.g. performance optimization, internal HW failure, etc.

# Logical/Physical Block Addressing

The mapping from LBA to physical block address can change over time.

- Internal HW failure.
- SSD writes in a funny way.
  - You have to erase before writing.
  - So, the SSD (for performance)
    - Writes to an empty block.
    - Erase the original block.
- Performance optimization on HDD
  - Based on block access patterns.
  - Place blocks on cylinder/sector/head in a way to minimize:
    - Seek
    - Rotate

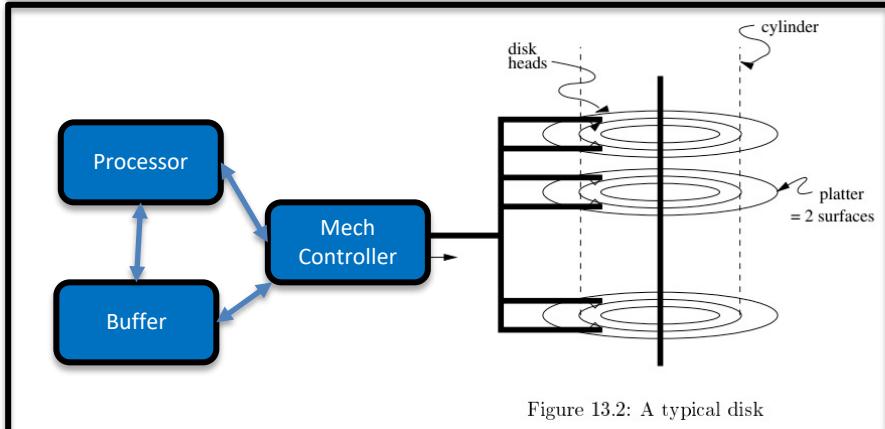
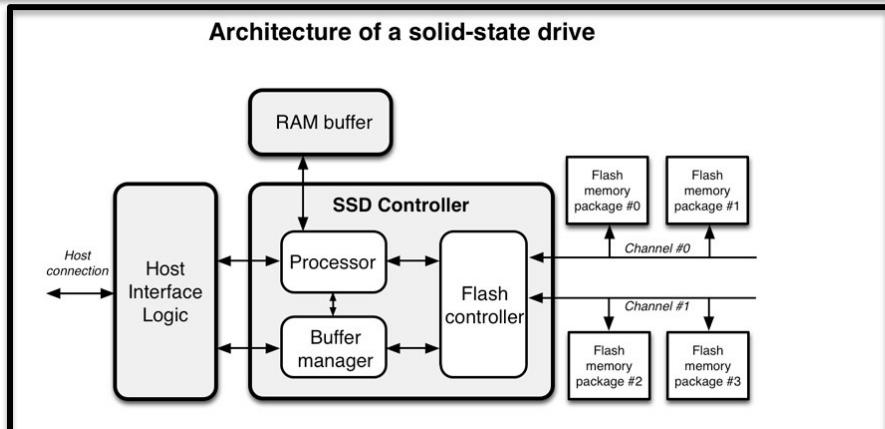
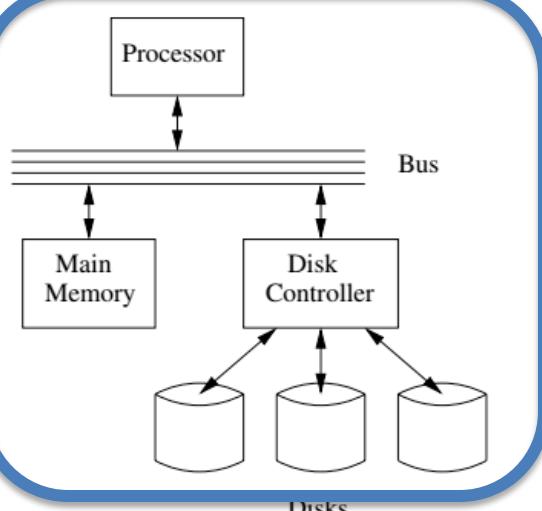


Figure 13.2: A typical disk

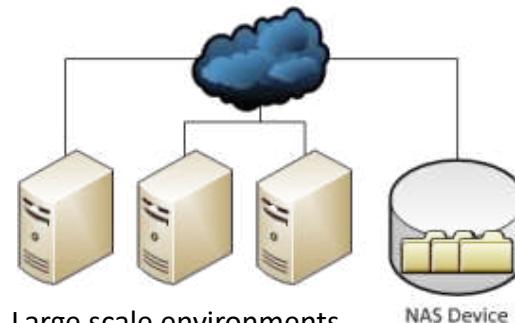
# I/O Architecture



How we normally think of disks and I/O.

## Network Attached Storage

- Shared storage over shared network
- File system
- Easier management

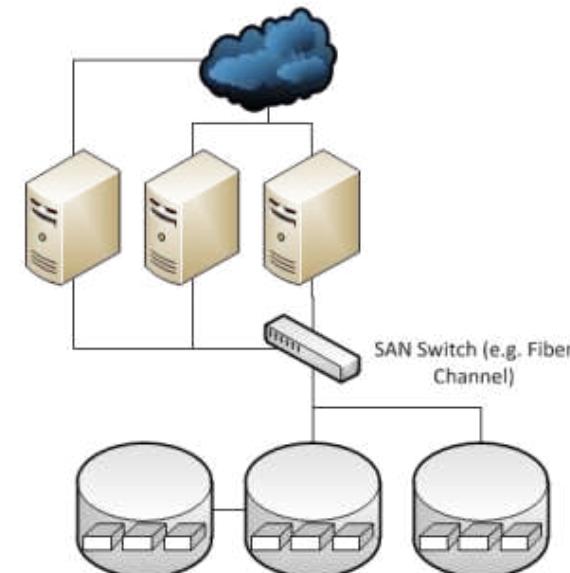


Large scale environments

- The bus-controller connection is over some kind of network.
- The disk controller is at the disks, and basically a “computer” with SW.
- Network is either
  - Standard communication network, or
  - Highly optimized I/O network.

## Storage Area Network

- Shared storage over dedicated network
- Raw storage
- Fast, but costly





# Magnetic Disks

- **Read-write head**
- Surface of platter divided into circular **tracks**
  - Over 50K-100K tracks per platter on typical hard disks
- Each track is divided into **sectors**.
  - A sector is the smallest unit of data that can be read or written.
  - Sector size typically 512 bytes
  - Typical sectors per track: 500 to 1000 (on inner tracks) to 1000 to 2000 (on outer tracks)
- To read/write a sector
  - disk arm swings to position head on right track
  - platter spins continually; data is read/written as sector passes under head
- Head-disk assemblies
  - multiple disk platters on a single spindle (1 to 5 usually)
  - one head per platter, mounted on a common arm.
- **Cylinder  $i$**  consists of  $i^{\text{th}}$  track of all the platters



# Magnetic Disks (Cont.)

- **Disk controller** – interfaces between the computer system and the disk drive hardware.
  - accepts high-level commands to read or write a sector
  - initiates actions such as moving the disk arm to the right track and actually reading or writing the data
  - Computes and attaches **checksums** to each sector to verify that data is read back correctly
    - If data is corrupted, with very high probability stored checksum won't match recomputed checksum
  - Ensures successful writing by reading back sector after writing it
  - Performs **remapping of bad sectors**



# Performance Measures of Disks

- **Access time** – the time it takes from when a read or write request is issued to when data transfer begins. Consists of:
  - **Seek time** – time it takes to reposition the arm over the correct track.
    - Average seek time is 1/2 the worst case seek time.
      - Would be 1/3 if all tracks had the same number of sectors, and we ignore the time to start and stop arm movement
    - 4 to 10 milliseconds on typical disks
  - **Rotational latency** – time it takes for the sector to be accessed to appear under the head.
    - 4 to 11 milliseconds on typical disks (5400 to 15000 r.p.m.)
    - Average latency is 1/2 of the above latency.
  - Overall latency is 5 to 20 msec depending on disk model
- **Data-transfer rate** – the rate at which data can be retrieved from or stored to the disk.
  - 25 to 200 MB per second max rate, lower for inner tracks



# Performance Measures (Cont.)

- **Disk block** is a logical unit for storage allocation and retrieval
  - 4 to 16 kilobytes typically
    - Smaller blocks: more transfers from disk
    - Larger blocks: more space wasted due to partially filled blocks
- **Sequential access pattern**
  - Successive requests are for successive disk blocks
  - Disk seek required only for first block
- **Random access pattern**
  - Successive requests are for blocks that can be anywhere on disk
  - Each access requires a seek
  - Transfer rates are low since a lot of time is wasted in seeks
- **I/O operations per second (IOPS)**
  - Number of random block reads that a disk can support per second
  - 50 to 200 IOPS on current generation magnetic disks



## Performance Measures (Cont.)

- **Mean time to failure (MTTF)** – the average time the disk is expected to run continuously without any failure.
  - Typically 3 to 5 years
  - Probability of failure of new disks is quite low, corresponding to a “theoretical MTTF” of 500,000 to 1,200,000 hours for a new disk
    - E.g., an MTTF of 1,200,000 hours for a new disk means that given 1000 relatively new disks, on an average one will fail every 1200 hours
  - MTTF decreases as disk ages

# Performance Measures of Disks

- **Access time** – the time it takes from when a read or write request is issued to when data transfer begins. Consists of:
  - **seek time** – time it takes to reposition the arm over the correct track.
    - Average seek time is 1/2 the worst case seek time.
      - Would be 1/3 if all tracks had the same number of sectors, and we ignore the time to start and stop arm movement
    - 4 to 10 milliseconds on typical disks
  - **Rotational latency** – time it takes for the sector to be accessed to appear under the head.
    - 4 to 11 milliseconds on typical disks (5400 to 15000 r.p.m.)
    - Average latency is 1/2 of the above latency.
  - Overall latency is 5 to 20 msec depending on disk model
- **Data-transfer rate** – the rate at which data can be retrieved from or stored to the disk.
  - 25 to 200 MB per second max rate, lower for inner tracks

From: Database System Concepts, 7<sup>th</sup> Ed.

# Logical Block Addressing ([https://gerardnico.com/wiki/data\\_storage/lba](https://gerardnico.com/wiki/data_storage/lba))

## 3 - The LBA scheme

| LBA        | C | H | S |
|------------|---|---|---|
| 0          | 0 | 0 | 0 |
| 1          | 0 | 0 | 1 |
| 2          | 0 | 0 | 2 |
| 3          | 0 | 0 | 3 |
| 4          | 0 | 0 | 4 |
| 5          | 0 | 0 | 5 |
| 6          | 0 | 0 | 6 |
| 7          | 0 | 0 | 7 |
| 8          | 0 | 0 | 8 |
| 9          | 0 | 0 | 9 |
| 10         | 0 | 1 | 0 |
| 11         | 0 | 1 | 1 |
| 12         | 0 | 1 | 2 |
| 13         | 0 | 1 | 3 |
| 14         | 0 | 1 | 4 |
| 15         | 0 | 1 | 5 |
| 16         | 0 | 1 | 6 |
| 17         | 0 | 1 | 7 |
| 18         | 0 | 1 | 8 |
| 19         | 0 | 1 | 9 |
| Cylinder 0 |   |   |   |

| LBA        | C | H | S |
|------------|---|---|---|
| 20         | 1 | 0 | 0 |
| 21         | 1 | 0 | 1 |
| 22         | 1 | 0 | 2 |
| 23         | 1 | 0 | 3 |
| 24         | 1 | 0 | 4 |
| 25         | 1 | 0 | 5 |
| 26         | 1 | 0 | 6 |
| 27         | 1 | 0 | 7 |
| 28         | 1 | 0 | 8 |
| 29         | 1 | 0 | 9 |
| 30         | 1 | 1 | 0 |
| 31         | 1 | 1 | 1 |
| 32         | 1 | 1 | 2 |
| 33         | 1 | 1 | 3 |
| 34         | 1 | 1 | 4 |
| 35         | 1 | 1 | 5 |
| 36         | 1 | 1 | 6 |
| 37         | 1 | 1 | 7 |
| 38         | 1 | 1 | 8 |
| 39         | 1 | 1 | 9 |
| Cylinder 1 |   |   |   |

lowing formula:

tor number

## Redundant Array of Independent Disks (RAID)



“RAID (redundant array of independent disks) is a data storage virtualization technology that combines multiple physical disk drive components into a single logical unit for the purposes of data redundancy, performance improvement, or both. (...)

RAID 0 consists of striping, without mirroring or parity. (...)

RAID 1 consists of data mirroring, without parity or striping. (...)

RAID 2 consists of bit-level striping with dedicated Hamming-code parity. (...)

RAID 3 consists of byte-level striping with dedicated parity. (...)

RAID 4 consists of block-level striping with dedicated parity. (...)

RAID 5 consists of block-level striping with distributed parity. (...)

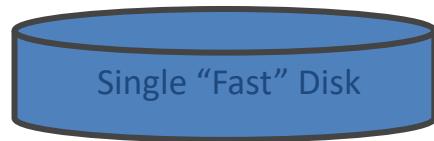
RAID 6 consists of block-level striping with double distributed parity. (...)

## Nested RAID

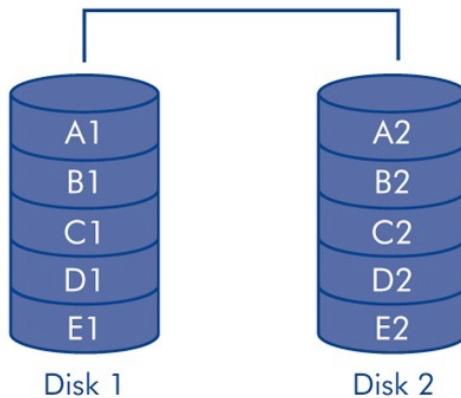
- RAID 0+1: creates two stripes and mirrors them. (...)
- RAID 1+0: creates a striped set from a series of mirrored drives. (...)
- **JBOD RAID N+N**: With JBOD (*just a bunch of disks*), (...)"

# RAID-0 and RAID-1

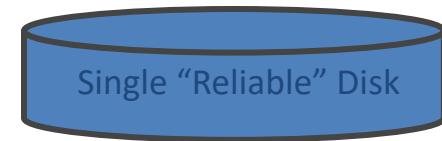
Two physical disks make  
one single, logical **fast** disk



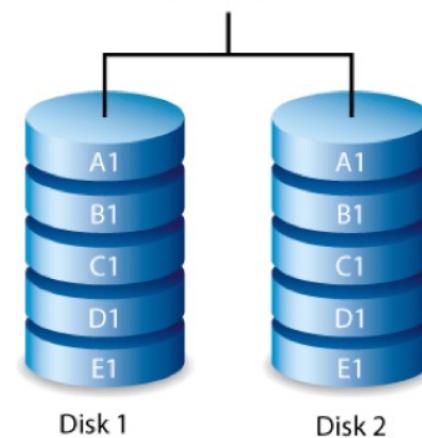
**RAID 0**



Two physical disks make  
one single, logical **reliable** disk

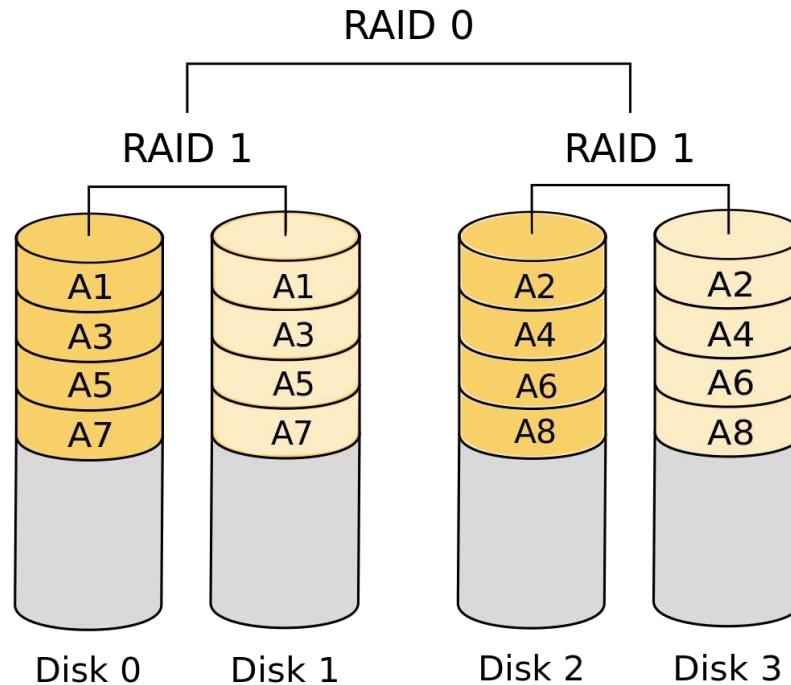


**RAID 1**



# Mixed RAID Modes

## RAID 1+0



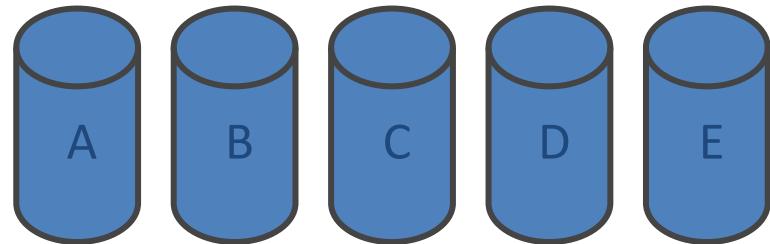
Stripe  
And  
Mirror

# RAID-5

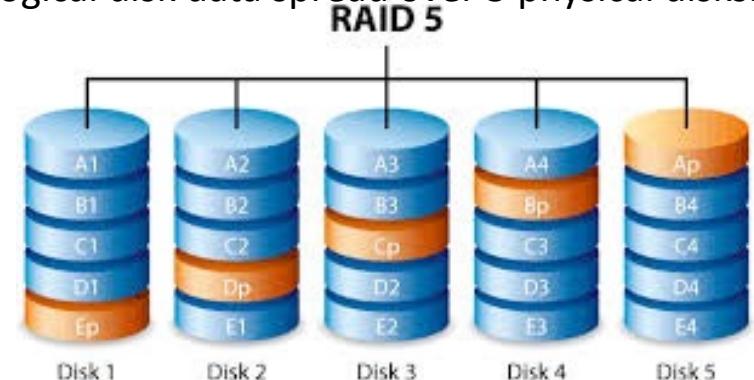
- Improved performance through parallelism
  - Rotation/seek
  - Transfer
- Availability uses *parity blocks*
  - Suppose I have 4 different data blocks on the logical drive A: A1, A2, A3, A4.
  - Parity function:  $A_p = P(A_1, A_2, A_3, A_4)$
  - Recovery function:  $A_2 = R(A_p, A_1, A_3, A_4)$
- During normal operations:
  - Read processing simply retrieves block.
  - Write processing of A2 updates A2 and Ap
- If an individual disk fails, the RAID
  - Read
    - Continues to function for reads on non-missing blocks.
    - Implements read on missing block by recalculating value.
  - Write
    - Updates block and parity block for non-missing blocks.
    - Computes missing block, and calculates parity based on old and new value.
  - Over time
    - “Hot Swap” the failed disk.
    - Rebuild the missing data from values and parity.



Is actually 5 smaller “logical” disks.

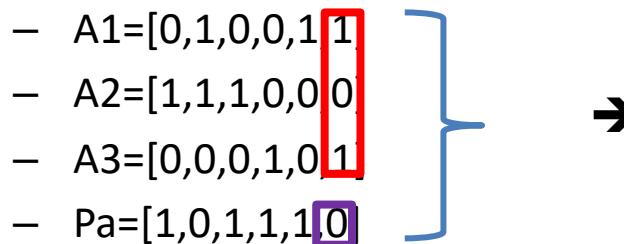


Logical disk data spread over 5 physical disks.



# Very Simple Parity Example

- Even-Odd Parity
  - $b[i]$  is an array of bits (0 or 1)
  - $P(b[i]) =$ 
    - 0 if an even number of bits = 1.  $\{P([0,1,1,0,1,1])=0\}$
    - 1 if an odd number of bits = 1.  $\{P(0,0,1,0,1,1)=1\}$
  - Given an array with one missing bit and the parity bit, I can re-compute the missing bit.
    - Case 1:  $[0,?,1,0,1,1]$  has  $P=0$ . There must be an EVEN number of ones and  $?=1$ .
    - Case 2:  $[0,?,1,0,1,1]$  has  $P=1$ . There must be an ODD number of ones and  $?=0$ .
- Block Parity applies this to a set of blocks bitwise



If I am missing a block and have the parity block, I can re-compute the missing block bitwise from remaining blocks and parity block.

# *NoSQL*

# *Concepts*

# Overview (I) (<https://en.wikipedia.org/wiki/NoSQL>)

A **NoSQL** (originally referring to "non SQL" or "non relational")<sup>[1]</sup> database provides a mechanism for storage and retrieval of data that is modeled in **means other than the tabular relations used in relational databases**. Such databases have existed since the late 1960s, but did not obtain the "NoSQL" moniker until a surge of popularity in the early twenty-first century,<sup>[2]</sup> triggered by the needs of Web 2.0 companies such as Facebook, Google, and Amazon.com.<sup>[3][4][5]</sup> NoSQL databases are increasingly used in big data and real-time web applications.<sup>[6]</sup> NoSQL systems are also sometimes called "**Not only SQL**" to emphasize that they may support SQL-like query languages.<sup>[7][8]</sup>

Motivations for this approach include: simplicity of design, simpler "horizontal scaling" to clusters of machines (which is a problem for relational databases),<sup>[2]</sup> and finer control over availability. The data structures used by NoSQL databases (e.g. key-value, wide column, graph, or document) are different from those used by default in relational databases, making **some operations faster in NoSQL**. The particular suitability of a given NoSQL database depends on the problem it must solve. Sometimes the **data structures used by NoSQL databases are also viewed as "more flexible"** than relational database tables.<sup>[9]</sup>

# Overview (I) (<https://en.wikipedia.org/wiki/NoSQL>)

Many NoSQL stores compromise [consistency](#) (in the sense of the [CAP theorem](#)) in favor of availability, partition tolerance, and speed. Barriers to the greater adoption of NoSQL stores include the use of low-level query languages (instead of SQL, for instance the lack of ability to perform ad-hoc joins across tables), lack of standardized interfaces, and huge previous investments in existing relational databases.<sup>[10]</sup> Most NoSQL stores lack true [ACID](#) transactions, ... ...

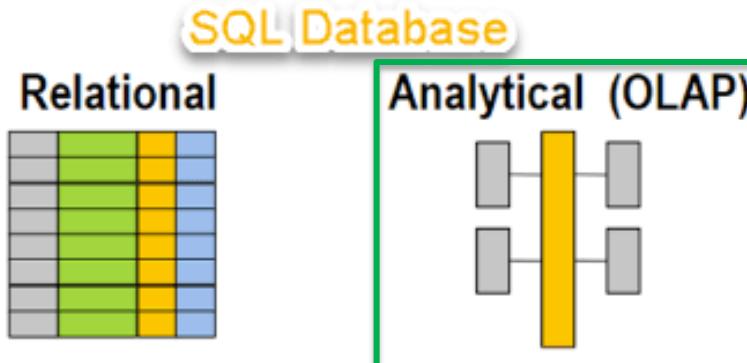
Instead, most NoSQL databases offer a concept of "eventual consistency" in which database changes are propagated to all nodes "eventually" (typically within milliseconds) so queries for data might not return updated data immediately or might result in reading data that is not accurate, a problem known as stale reads.<sup>[11]</sup> Additionally, some NoSQL systems may exhibit lost writes and other forms of [data loss](#).<sup>[12]</sup> Fortunately, some NoSQL systems provide concepts such as [write-ahead logging](#) to avoid data loss.<sup>[13]</sup> For [distributed transaction processing](#) across multiple databases, data consistency is an even bigger challenge that is difficult for both NoSQL and relational databases. Even current relational databases "do not allow referential integrity constraints to span databases."<sup>[14]</sup>

# Simplistic Classification

(<https://medium.com/swlh/4-types-of-nosql-databases-d88ad21f7d3b>)

Relational is the foundational model.

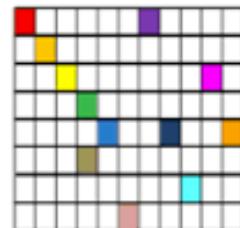
We covered graphs and examples.



We will see OLAP in a future lecture.

Subject of this lecture and part of HW3

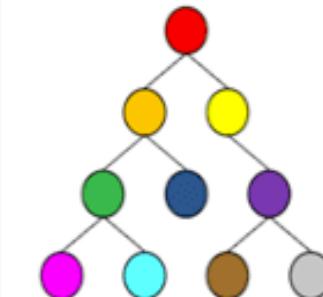
Column-Family



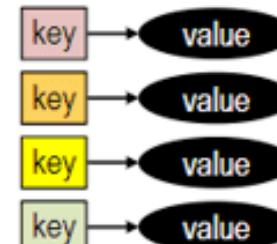
Graph



Document



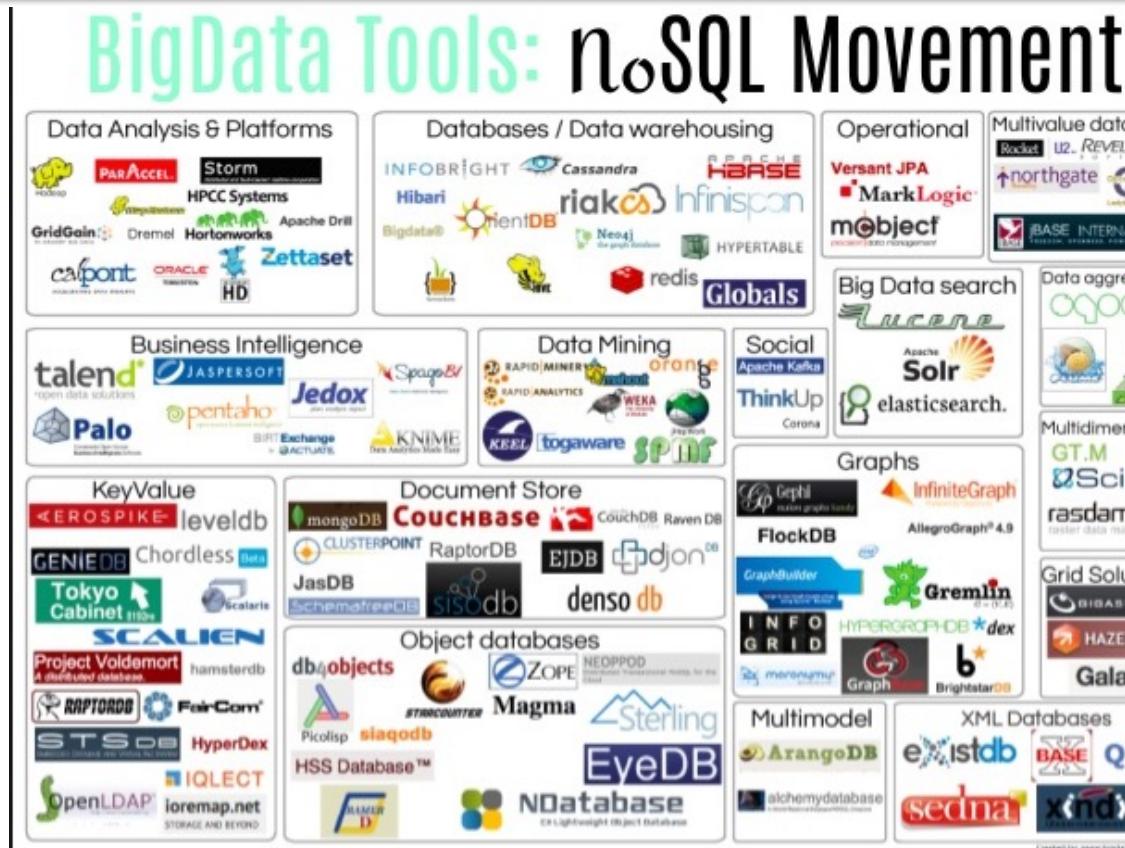
Key-Value



# One Taxonomy

| Document Database                                                                                                                                                                                                                                                                                                                          | Graph Databases                                                                                                                                                                                                                                                                                                                                                                                                                        |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|  <br>  |  <br>The Distributed Graph Database                                                                                                                                                                                                                                |
| Wide Column Stores                                                                                                                                                                                                                                                                                                                         | Key-Value Databases                                                                                                                                                                                                                                                                                                                                                                                                                    |
|  <br>  |  <br> <br> |

# Another Taxonomy



# Use Cases

## Motivations

- Massive write performance.
- Fast key value look ups.
- Flexible schema and data types.
- No single point of failure.
- Fast prototyping and development.
- Out of the box scalability.
- Easy maintenance.

## What is wrong with SQL/Relational?

- Nothing. One size fits all? Not really.
- Impedance mismatch. – Object Relational Mapping doesn't work quite well.
- Rigid schema design.
- Harder to scale.
- Replication.
- Joins across multiple nodes? Hard.
- How does RDMS handle data growth? Hard.
- Need for a DBA.
- Many programmers are already familiar with it.
- Transactions and ACID make development easy.
- Lots of tools to use.

# *Documents*

# Example Document – An Order

## FOOD ORDER FORM TEMPLATE

Company Name  
123 Main Street  
Hamilton, OH 44416  
(321) 456-7890  
Email Address  
Point of Contact  
web address

YOUR LOGO

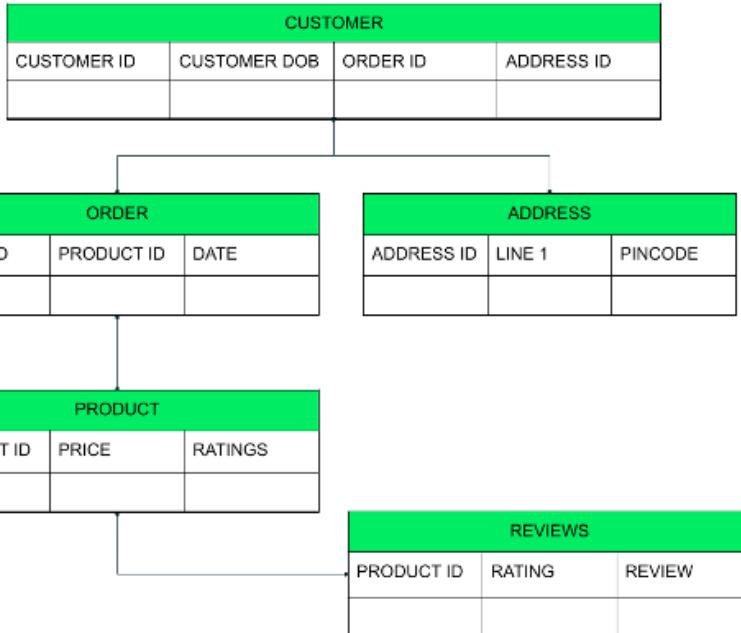
## ORDER FORM

| CUSTOMER                  |  | ORDER NO.         | ORDER DATE  |
|---------------------------|--|-------------------|-------------|
| ATTN: Name / Dept         |  | DATE NEEDED       | TIME NEEDED |
| Company Name              |  | ORDER RECEIVED BY |             |
| 123 Main Street           |  |                   |             |
| Hamilton, OH 44416        |  |                   |             |
| (321) 456-7890            |  |                   |             |
| Email Address             |  |                   |             |
| <b>DESCRIPTION</b>        |  |                   |             |
|                           |  | UNIT PRICE        | AMOUNT      |
|                           |  | \$ -              |             |
|                           |  | \$ -              |             |
|                           |  | \$ -              |             |
|                           |  | \$ -              |             |
|                           |  | \$ -              |             |
|                           |  | \$ -              |             |
|                           |  | \$ -              |             |
|                           |  | \$ -              |             |
|                           |  | \$ -              |             |
| <b>FINANCIALS</b>         |  |                   |             |
| enter percentage          |  | TAX RATE          | 0.000%      |
| enter initial pymt amount |  | TOTAL TAX         | \$ -        |
|                           |  | DELIVERY          | \$ -        |
|                           |  | GRAND TOTAL       | \$ -        |
|                           |  | LESS PAYMENT      | \$ -        |
| THANK YOU!                |  |                   |             |

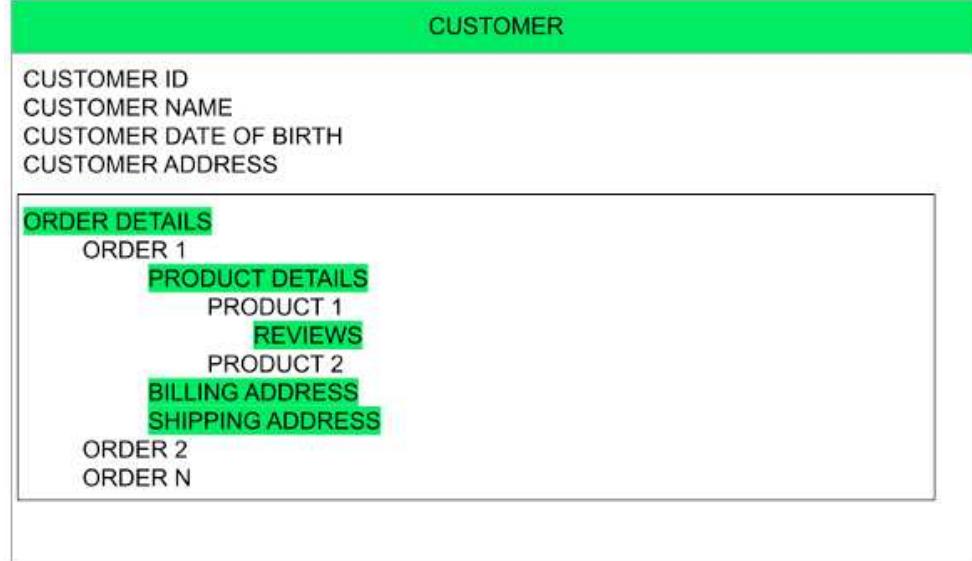
- There are 5 entity types on the form:
  - A “copy of” customer information.
  - Links (via productCode) to products
  - Order
  - OrderDetails
  - Comments
- But OrderDetails and Comments are somehow different from the others.
  - These are arrays of objects
  - That are sort of “inside” the order.
- These are weak entities, but relational does not always handle these well.

# Relational vs Document

Relational



Relational



The key difference?

- Relational DB attributes are atomic.
- Document DB attributes may be *multi-valued* and *complex*.

# UI Interface and Logical Data Model

Columbia University

|                                                                                        |                 |            |               |              |                                       |
|----------------------------------------------------------------------------------------|-----------------|------------|---------------|--------------|---------------------------------------|
| CUSTOMER NUMBER                                                                        | 10001           |            |               |              |                                       |
| CONTACT NAME                                                                           | CONTACT NUMBER  |            |               |              |                                       |
| Lord Voldemort                                                                         | +1 212-555-6666 |            |               |              |                                       |
| Customer Address (THIS WAS INSANELY PAINFUL TO LAYOUT)                                 |                 |            |               |              |                                       |
| STREET 1                                                                               | STREET 2        |            |               |              |                                       |
| 520 W 120th St.                                                                        | 4th Floor       |            |               |              |                                       |
| CITY                                                                                   | STATE           | COUNTRY    | POSTAL CODE   |              |                                       |
| New York                                                                               | NY              | US         | 10027         |              |                                       |
| <b>I nearly went "postal" doing the previous layout and almost failed all of you!"</b> |                 |            |               |              |                                       |
| NUMBER                                                                                 | STATUS          | ORDER DATE | REQUIRED DATE | SHIPPED DATE | DETAILS                               |
| 21                                                                                     | Shipped         | 2022-01-01 | 2022-02-01    | 2022-01-28   | <input type="button" value="Expand"/> |
| LINE NUMBER                                                                            | PRODUCT CODE    | QUANTITY   | PRICE EACH    |              |                                       |
| 0                                                                                      | P0              | 0          | 0             |              |                                       |
| 1                                                                                      | P1              | 2          | 3             |              |                                       |
| 2                                                                                      | P2              | 4          | 6             |              |                                       |
| NUMBER                                                                                 | STATUS          | ORDER DATE | REQUIRED DATE | SHIPPED DATE | DETAILS                               |
| 22                                                                                     | Pending         | 2022-03-01 | 2022-04-01    | 0000000      | <input type="button" value="Expand"/> |

Note to DFF: Start

- [~/Dropbox/000/000-A-Current-Examples/W4111-FastAPI-IMDB-Solution](#)
- [~/Dropbox/000/000-A-Current-Examples/current-dashboard](#)

```
{
 customerName: "Columbia University",
 contact: {
 name: "Lord Voldemort",
 phone: "+1 212-555-6666"
 },
 address: {
 line1: "520 W 120th St."
 line2: "Floor 4",

 },
 orders: [
 {
 number: ...,

 details: [

]
 }
]
}
```

# Switch to Notebook



# *MongoDB*

# MongoDB Concepts

| RDBMS                                    | MongoDB                                                               |
|------------------------------------------|-----------------------------------------------------------------------|
| Database                                 | Database                                                              |
| Table                                    | Collection                                                            |
| Tuple/Row                                | Document                                                              |
| column                                   | Field                                                                 |
| Table Join                               | Embedded Documents                                                    |
| Primary Key                              | Primary Key (Default key <code>_id</code> provided by MongoDB itself) |
| Database Server, Client, Tools, Packages |                                                                       |
| mysqld/Oracle                            | <code>mongod</code>                                                   |
| mysql/sqlplus                            | <code>mongo</code>                                                    |
| DataGrip                                 | Compass                                                               |
| pymysql                                  | <code>pymongo</code>                                                  |

# Core Operations

## Basic Operations:

- Create database
- Create collection
- Create-Retrieve-Update-Delete (CRUD):
  - Create: insert()
  - Retrieve:
    - find()
    - find\_one()
  - Update: update()
  - Delete: remove()

## More Advanced Concepts:

- Limit
- Sort
- Aggregation Pipelines
  - Merge
  - Union
  - Lookup
  - Match
  - Merge
  - Sample
  - ... ...

We will just cover the basics for now and may cover more things in HW or other lectures.

# find()

- Note:
  - MongoDB uses a more `pymysql` approach, e.g. an API, than pure declarative languages like SQL.
  - The parameters for `find()` are where the declarative language appears.
- The basic forms of `find()` and `find_one()` have two parameters:
  - *filter expression*
  - *Project expression*
- You can use the Compass tool and screen captures for some HW and exam answers.
- What if I want the answer in a Jupyter Notebook?

The screenshot shows the MongoDB Compass interface for the `GOT.seasons` collection. The `Documents` tab is active. The search bar contains the query `{"episodes.scenes.location": "The Dothraki Sea"}`. Below the search bar, there are filter, project, sort, and collation options. The results pane displays three documents. Each document has an `_id`, `season` (set to 1), and an `episodes` array. The first document's `episodes` array is expanded to show 10 individual episode objects, each with `episodeNum`, `episodeTitle`, and `episodeLink`.

```
_id: ObjectId("60577e50c68b67110968b6d1")
season: "1"
episodes: Array
 ▾ 0: Object
 episodeNum: 1
 episodeTitle: "Winter Is Coming"
 episodeLink: "/title/tt1480055/"
 ▾ 1: Object
 ▾ 2: Object
 ▾ 3: Object
 ▾ 4: Object
 ▾ 5: Object
 ▾ 6: Object
 ▾ 7: Object
 ▾ 8: Object
 ▾ 9: Object

_id: ObjectId("60577e50c68b67110968b6d5")
season: "5"
episodes: Array

_id: ObjectId("60577e50c68b67110968b6d6")
season: "5"
episodes: Array
```

# Generate Code

The screenshot shows the MongoDB Compass interface. On the left, the sidebar lists databases (Local, 7 DBs) and collections (6 COLLECTIONS). The 'GOT.seasons' collection is selected. In the main area, a document is displayed with fields: \_id, season, and episodes. Below the document, two results are shown, each with an \_id, season, and episodes field. A modal window titled 'Export Query To Language' is open. It contains a 'My Query:' section with the following MongoDB query:

```
1 {
2 "episodes.scenes.location": "The Dothraki Sea"
3 }
```

Below this, the 'Export Query To:' dropdown is set to 'PYTHON 3'. The generated Python code is:

```
1 # Requires the PyMongo package.
2 # https://api.mongodb.com/python/current
3
4 client = MongoClient('mongodb://localhost:27017/?
5 filter={
6 'episodes.scenes.location': 'The Dothraki Sea'
7 }
8 project={
9 'season': 1,
10 'episodes.episodeNum': 1,
11 'episodes.episodeLink': 1,
12 'episodes.episodeTitle': 1
13 }
14
15 result = client['GOT']['seasons'].find(
16 filter=filter,
```

At the bottom of the modal, there are two checkboxes: 'Include Import Statements' (unchecked) and 'Include Driver Syntax' (checked). A 'CLOSE' button is at the bottom right.

- Choose Export to Language.
- Copy into the notebook.
- The export has an option to include all the connection setup, choosing DB, ... ...
- Switch to Notebook

# Result is not Quite You Expect

GOT.seasons

DOCUMENTS 8 TOTAL SIZE 1.0MB AVG. SIZE 130.2KB INDEXES 1 TOTAL SIZE 20.0KB AVG. SIZE 20.0KB

Documents Aggregations Schema Explain Plan Indexes Validation

FILTER: {"episodes.scenes.location": "The Dothraki Sea"}  
PROJECT: { season: 1, "episodes.episodeNum":1, "episodes.episodeLink": 1, "episodes.episodeTitle": 1}  
SORT: { field: -1 }  
COLLATION: { locale: 'simple' }

FIND RESET ...  
MAX TIME MS: 60000  
SKIP: 0 LIMIT: 0

VIEW Refresh

Displaying documents 1 - 3 of 3 < > C REFRESH

```
_id:ObjectId("60577e50c68b67110968b6d1")
season:"1"
episodes:Array
 ▾ 0:Object
 episodeNum: 1
 episodeTitle: "Winter Is Coming"
 episodeLink: "/title/tt1480055/"
 ▾ 1:Object
 ▾ 2:Object
 ▾ 3:Object
 episodeNum: 4
 episodeTitle: "Cripples, Bastards, and Broken Things"
 episodeLink: "/title/tt1829963/"
 ▾ 4:Object
 ▾ 5:Object
 ▾ 6:Object
 ▾ 7:Object
 ▾ 8:Object
 ▾ 9:Object

_id:ObjectId("60577e50c68b67110968b6d5")
season:"5"
episodes:Array

_id:ObjectId("60577e50c68b67110968b6d6")
season:"6"
episodes:Array
```

- The query returns documents that match.
  - The document is “Large” and has seasons and episodes and seasons.
  - If you do a \$project requesting episodes/episode content,
    - You get all episodes in the documents that match.
    - Not just the episodes with the scene/location.
    - Projecting array elements from arrays whose elements are arrays is complex and baffling.
- You also get back something (a cursor) that is iterable.

# Result is not Quite You Expect

The screenshot shows the MongoDB Compass interface. At the top, there's a navigation bar with tabs for 'Documents', 'Aggregations', 'Schema', 'Explain Plan', 'Indexes', and 'Validation'. Below the navigation bar, there are sections for 'FILTER', 'PROJECT', 'SORT', and 'COLLATION' with their respective JSON configurations. The 'FILTER' section contains the query: { "episodes.scenes.location": "The Dothraki Sea" }. The 'PROJECT' section contains the projection: { season: 1, "episodes.episodeNum": 1, "episodes.episodeLink": 1, "episodes.episodeTitle": 1 }. The 'SORT' section has a sort key of { field: -1 }. The 'COLLATION' section specifies a locale of 'simple'. On the right side of the interface, there are buttons for 'OPTIONS', 'FIND', 'RESET', and '...', along with 'MAX TIME MS' set to 60000, 'SKIP' set to 0, and 'LIMIT' set to 0. Below the configuration, there's a 'VIEW' button followed by four small icons. The main area displays the results of the query as a grid of documents. Each document has an '\_id' field, a 'season' field (either '5' or '6'), and an 'episodes' array. The 'episodes' array contains multiple objects, each with an 'episodeNum' field (ranging from 0 to 9) and an 'episodeLink' field. The 'episodeTitle' field is also present in some documents. The JSON output is shown in a tree-view format where arrays are expanded to show individual elements.

Net for HWs and exams:

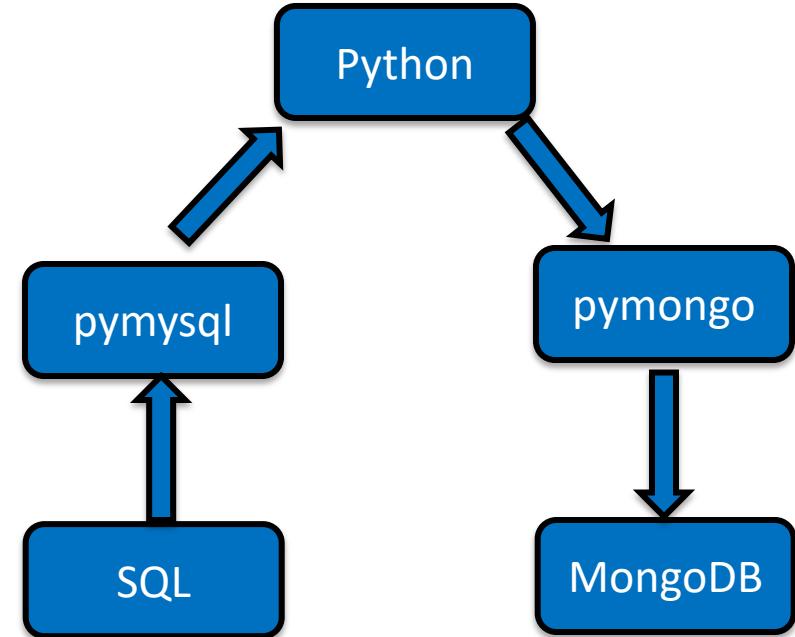
- We will keep the queries simple.
- The language is as complex as SQL, and we spent several weeks on the language.
- Let's take a look in the Jupyter Notebook on some create and insert functions.
- But, first, the datamodel impedance mismatch concept.

- The query returns documents that match.
  - The document is “Large” and has many episodes and seasons.
    - You do a \$project requesting episodes/episode content,
      - You get all episodes in the documents that match.
      - Not just the episodes with the scene/location.
      - Projecting array elements from arrays whose elements are arrays is complex and baffling.
    - You also get back something (a cursor) that is iterable.

# More Fun – Data Types

MongoDB supports many datatypes. Some of them are –

- **String** – This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.
- **Integer** – This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
- **Boolean** – This type is used to store a boolean (true/ false) value.
- **Double** – This type is used to store floating point values.
- **Min/ Max keys** – This type is used to compare a value against the lowest and highest BSON elements.
- **Arrays** – This type is used to store arrays or list or multiple values into one key.
- **Timestamp** – timestamp. This can be handy for recording when a document has been modified or added.
- **Object** – This datatype is used for embedded documents.
- **Null** – This type is used to store a Null value.
- **Symbol** – This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.
- **Date** – This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
- **Object ID** – This datatype is used to store the document's ID.
- **Binary data** – This datatype is used to store binary data.
- **Code** – This datatype is used to store JavaScript code into the document.
- **Regular expression** – This datatype is used to store regular expression.



# (Some) MongoDB CRUD Operations

- Create:
  - db.collection.insertOne()
  - db.collection.insertMany()
- Retrieve:
  - db.collection.find()
  - db.collection.findOne()
  - db.collection.findOneAndUpdate()
  - ....
- Update:
  - db.collection.updateOne()
  - db.collection.updateMany()
  - db.collection.replaceOne()
- Delete:
  - db.collection.deleteOne()
  - db.collection.deleteMany()

pymongo maps the camel case to \_, e.g.

- findOne()
- find\_one()

There are good online tutorials:

- [https://www.tutorialspoint.com/python\\_data\\_access](https://www.tutorialspoint.com/python_data_access)
- <https://www.tutorialspoint.com/mongodb/index.htm>

# (Some) MongoDB Pipeline Operators

<https://www.slideshare.net/mongodb/s01-e04-analytics>

## Aggregation operators

- Pipeline and Expression operators

| Pipeline  | Expression | Arithmetic | Conditional |
|-----------|------------|------------|-------------|
| \$match   | \$addToSet | \$add      | \$cond      |
| \$sort    | \$first    | \$divide   | \$ifNull    |
| \$limit   | \$last     | \$mod      |             |
| \$skip    | \$max      | \$multiply |             |
| \$project | \$min      | \$subtract |             |
| \$unwind  | \$avg      |            | Variables   |
| \$group   | \$push     |            |             |
| \$geoNear | \$sum      |            |             |
| \$text    |            |            | \$let       |
| \$search  |            |            | \$map       |

Tip: Other operators for date, time, boolean and string manipulation

# MongoDB Checkpoint

- You can see that MongoDB has powerful, sophisticated
  - Operators
  - Expressions
  - Pipelines
- We have only skimmed the surface. There is a lot more:
  - Indexes
  - Replication, Sharding
  - Embedded Map-Reduce support
  - ... ...
- We will explore a little more in subsequent lectures, homework, ... ...
- You will have to install MongoDB and Compass for HW4 and final exam.

*W4111 – Introduction to Databases  
Section 002, Spring 2024*

*Lecture 2: ER, Relational, SQL (I)*



# *W4111 – Introduction to Databases*

## *Section 002, Spring 2024*

### *Lecture 2: ER, Relational, SQL (I)*

We will start in a couple of minutes.

# *Contents*

# Contents

- Introduction: Status updates
- ER (Diagram) Modeling (Chapter 6)
- The Relational Model and Algebra (Chapter 2)
- SQL (Chapter 3)

# *Introduction*

# Status Updates

- TAs:
  - The TA's are on-board, although we may be getting a couple more.
  - We had a kickoff meeting and begun the work.
  - The TA will be setting up HW 0 submission, which will be very simple.
  - The TA's have started posting OHs on the instructors' page of the website.  
[\(https://donald-f-ferguson.github.io/W4111-Intro-to-Databases-Spring-2024/instructors.html\)](https://donald-f-ferguson.github.io/W4111-Intro-to-Databases-Spring-2024/instructors.html)
- HW 0 is due Sunday (28-JAN) at 11:59 PM.
- HW 1 come out this weekend.
  - You will have two weeks.
  - There will two parts:
    - Written questions based on material from 1<sup>st</sup> lecture and slides associated with textbook.
    - Practical tasks.

# Database design, Entity-Relationship Model (Part 1)

# *Concepts*

## *(Reminder, Cover Quickly)*



# Design Phases

- Initial phase -- characterize fully the data needs of the prospective database users.
- Second phase -- choosing a data model
  - Applying the concepts of the chosen data model
  - Translating these requirements into a conceptual schema of the database.
  - A fully developed conceptual schema indicates the functional requirements of the enterprise.
    - Describe the kinds of operations (or transactions) that will be performed on the data.

## DFF Comments:

- We see slides with this formatting, they come directly from the presentations associated with the textbook. (<https://www.db-book.com/>)
- The number at the bottom is of the form chapter.slide\_no.
- I try to put my comments, modifications and annotations in red text, or inside a red rectangle/callout.



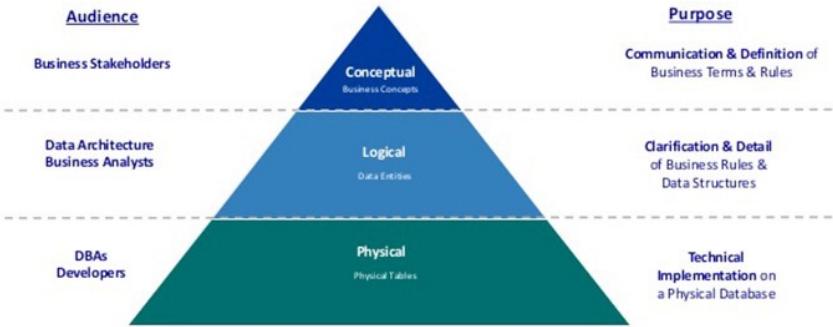
# Design Phases (Cont.)

- Final Phase -- Moving from an abstract data model to the implementation of the database
  - Logical Design – Deciding on the database schema. Database design requires that we find a “good” collection of relation schemas.
  - Business decision – What attributes should we record in the database?
  - Computer Science decision – What relation schemas should we have and how should the attributes be distributed among the various relation schemas?
  - Physical Design – Deciding on the physical layout of the database

# A Common and my Approach: Conceptual → Logical → Physical

<https://ehkioya.com/conceptual-logical-physical-database-modeling/>

## Levels of Data Modeling

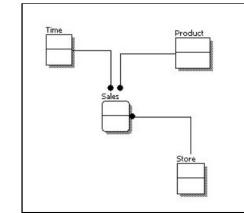


- It is easy to get carried away with modeling. You can spend all your time modeling and not actually build the schema.
- We will use the approaches in class.
- Mostly to understand concepts and patterns.

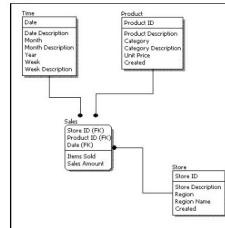
<https://www.1keydata.com/datawarehousing/data-modeling-levels.html>

| Feature              | Conceptual | Logical | Physical |
|----------------------|------------|---------|----------|
| Entity Names         | ✓          | ✓       |          |
| Entity Relationships | ✓          | ✓       |          |
| Attributes           |            | ✓       |          |
| Primary Keys         |            | ✓       | ✓        |
| Foreign Keys         |            | ✓       | ✓        |
| Table Names          |            |         | ✓        |
| Column Names         |            |         | ✓        |
| Column Data Types    |            |         | ✓        |

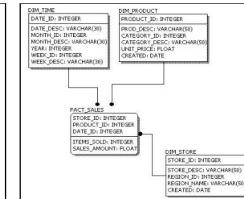
Conceptual Model Design



Logical Model Design



Physical Model Design



<https://www.1keydata.com/datawarehousing/data-modeling-levels.html>



# ER model -- Database Modeling

- The ER data mode was developed to facilitate database design by allowing specification of an **enterprise schema** that represents the overall logical structure of a database.
- The ER data model employs three basic concepts:
  - entity sets,
  - relationship sets,
  - attributes.
- The ER model also has an associated diagrammatic representation, the **ER diagram**, which can express the overall logical structure of a database graphically.



# Entity Sets

COMS W4111 002 01 2024

- An **entity** is an object that exists and is distinguishable from other objects.
  - Example: specific person, company, event, plant
- An **entity set** is a set of entities of the **same type** that share the same properties.
  - Example: set of all persons, companies, trees, holidays
- An entity is represented by a set of attributes; i.e., descriptive properties **possessed by all** members of an entity set.
  - Example:  
 $\text{instructor} = (\text{ID}, \text{name}, \text{salary})$   
 $\text{course} = (\text{course\_id}, \text{title}, \text{credits})$
- A subset of the attributes form a **primary key** of the entity set; i.e., uniquely identifying each member of the set.

## DFF Comments:

- Some of these statements apply primarily to OO systems and the relational/SQL models.
- A motivation for “No SQL” is to relax the constraints.



# Entity Sets -- *instructor* and *student*

|       |            |
|-------|------------|
| 76766 | Crick      |
| 45565 | Katz       |
| 10101 | Srinivasan |
| 98345 | Kim        |
| 76543 | Singh      |
| 22222 | Einstein   |

*instructor*

|       |         |
|-------|---------|
| 98988 | Tanaka  |
| 12345 | Shankar |
| 00128 | Zhang   |
| 76543 | Brown   |
| 76653 | Aoi     |
| 23121 | Chavez  |
| 44553 | Peltier |

*student*



# Representing Entity sets in ER Diagram

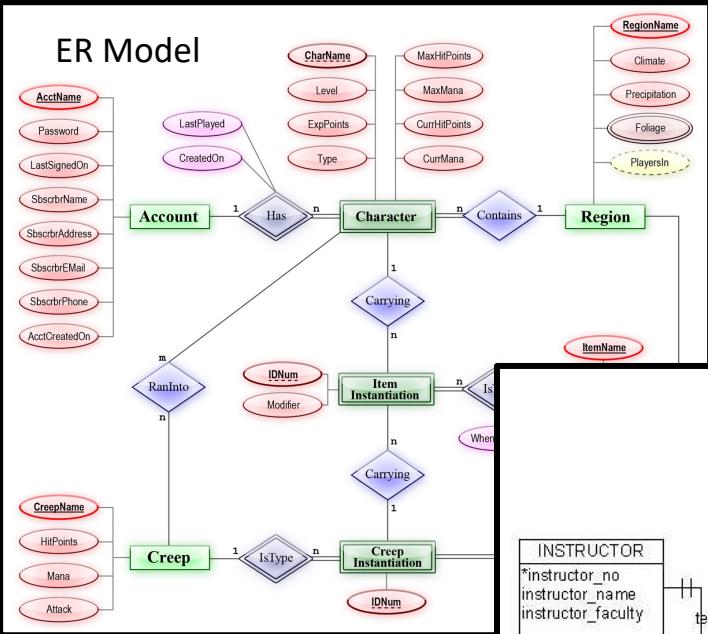
- Entity sets can be represented graphically as follows:
  - Rectangles represent entity sets.
  - Attributes listed inside entity rectangle
  - Underline indicates primary key attributes

| <i>instructor</i> |
|-------------------|
| <u>ID</u>         |
| <i>name</i>       |
| <i>salary</i>     |

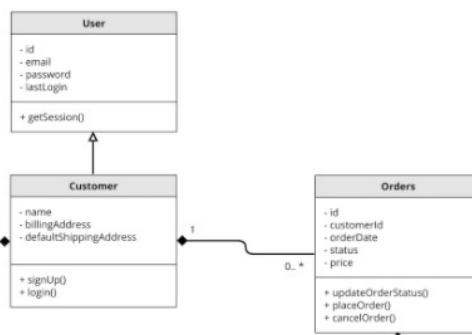
| <i>student</i>  |
|-----------------|
| <u>ID</u>       |
| <i>name</i>     |
| <i>tot_cred</i> |

# Visual Notation – Many Notations

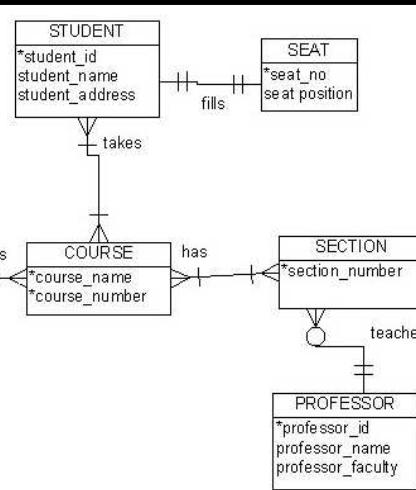
ER Model



UML



Crow's Foot



- “Other,” i.e. PowerPoint is the most common modeling notation.
- It is easy to get “carried away.”
- The trick is to do “just enough modeling.”
- I mostly use Crow’s Foot
  - It is “just enough”
  - But lacks some capabilities.
- The book uses ER notation.

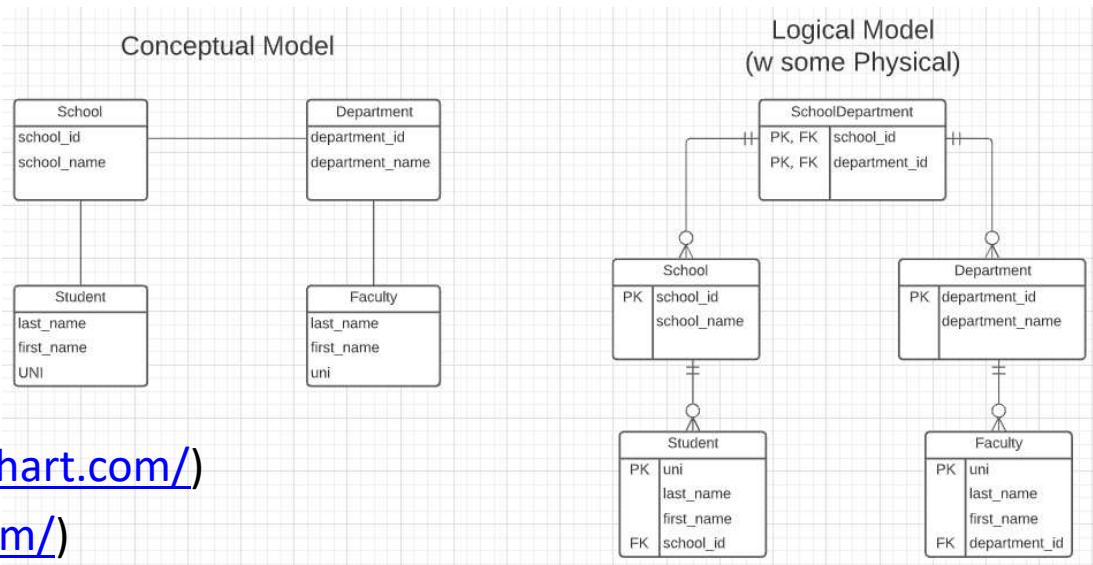
# First Iteration/Step for University Data Model

- This is the level of detail I want when I ask for:
  - Conceptual Model diagram.
  - Logical Model diagram.

- Some online tools with “free,” constrained usage.

- Lucidchart (<https://www.lucidchart.com/>)
- Vertabelo (<https://vertabelo.com/>)

- The model has:
  - Four entity sets: *School*, *Department*, *Student*, *Faculty*
  - Three relationship sets: *Student-School*, *School-Department*, *Faculty-Department*.





# Entity Sets -- *instructor* and *student*

|       |            |
|-------|------------|
| 76766 | Crick      |
| 45565 | Katz       |
| 10101 | Srinivasan |
| 98345 | Kim        |
| 76543 | Singh      |
| 22222 | Einstein   |

*instructor*

|       |         |
|-------|---------|
| 98988 | Tanaka  |
| 12345 | Shankar |
| 00128 | Zhang   |
| 76543 | Brown   |
| 76653 | Aoi     |
| 23121 | Chavez  |
| 44553 | Peltier |

*student*

DFF Comments: Just a reminder of a previous slide.



# Relationship Sets

- A **relationship** is an association among several entities

Example:

44553 (Peltier)                  advisor                  22222 (Einstein)  
student entity                  relationship set                  instructor entity

- A **relationship set** is a mathematical relation among  $n \geq 2$  entities, each taken from entity sets

$$\{(e_1, e_2, \dots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \dots, e_n \in E_n\}$$

where  $(e_1, e_2, \dots, e_n)$  is a relationship

- Example:

$$(44553, 22222) \in \text{advisor}$$

## DFF Comments:

- Nobody thinks about relationships this way.
- There is no idea so simple that a DB professor cannot make it confusing, usually by using math.



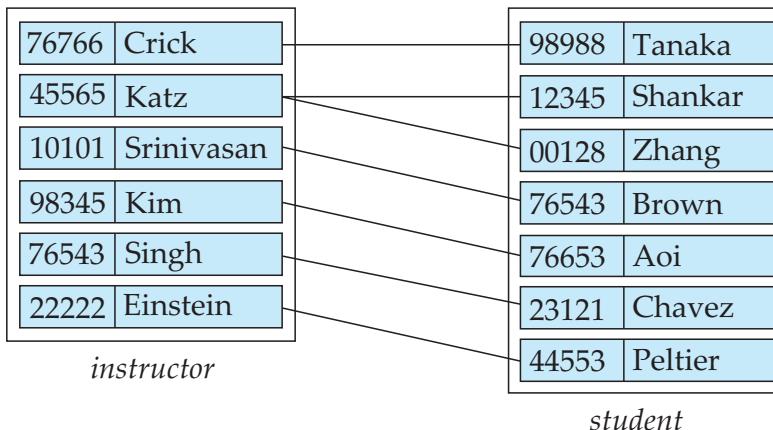
$\{(e_1, e_2) \mid e_1 \in \text{instructor}, e_2 \in \text{students}\}$

| Instructor | Student |
|------------|---------|
| 76766      | 98988   |
| 45565      | 12345   |
| 45565      | 00128   |
|            |         |
|            |         |



# Relationship Sets (Cont.)

- Example: we define the relationship set *advisor* to denote the associations between students and the instructors who act as their advisors.
- Pictorially, we draw a line between related entities.



| Faculty | Student |
|---------|---------|
| 76766   | 98988   |
| 45565   | 12345   |
| 45565   | 00128   |
|         |         |
|         |         |
|         |         |

DFF Comments:

- Nobody draws the diagrams this way, but ...
- Sometimes thinking this way helps understand other ways to depict the concept.

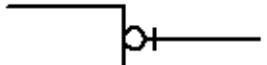
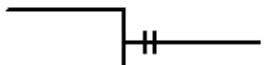
(dff9, W4111)

# Notation has Precise Meaning

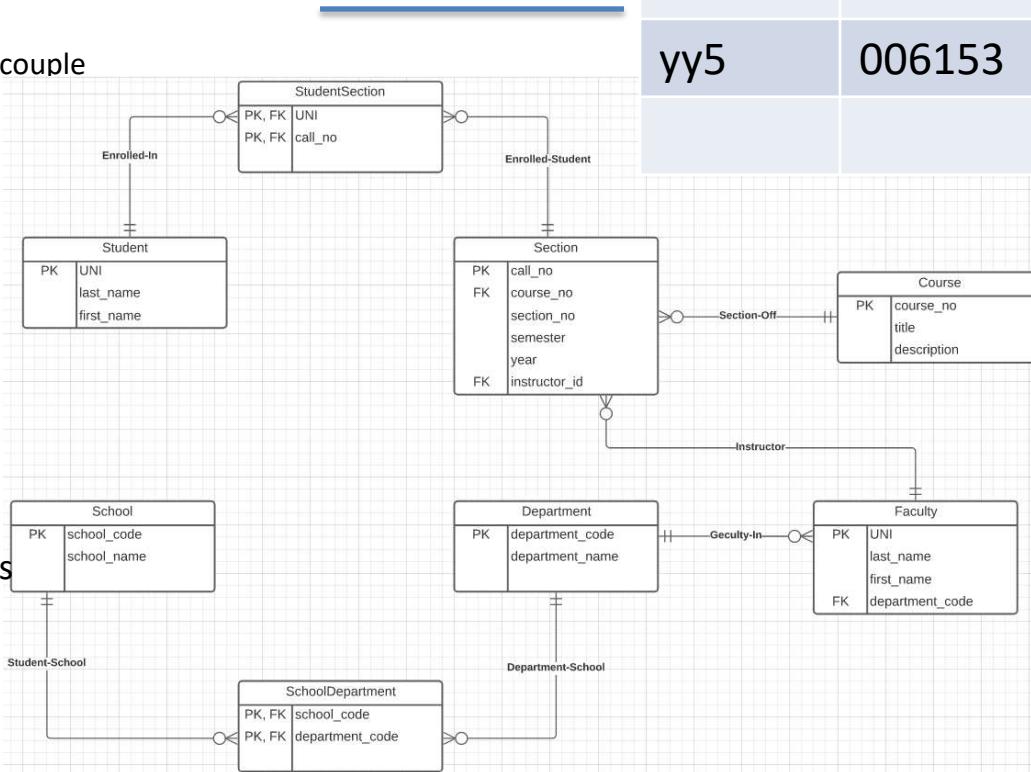
| UNI  | CALLNO |
|------|--------|
| dff9 | 006135 |
| dff9 | 110234 |
| yy5  | 006153 |

- Attribute annotations:
  - PK = Primary Key
  - FK = Foreign Key
- Line annotations:
  - We will spend a lot of time discussing keys.
  - We will start in a couple of slides.

## Summary of Crow's Foot Notation

-  One or Zero
-  One and only One
-  Zero or Many
-  One or Many

We will learn over time and there are good tutorials (<https://www.lucidchart.com/pages/er-diagrams>) to help study and refresh.



# What Does this Mean? Let's Get Started

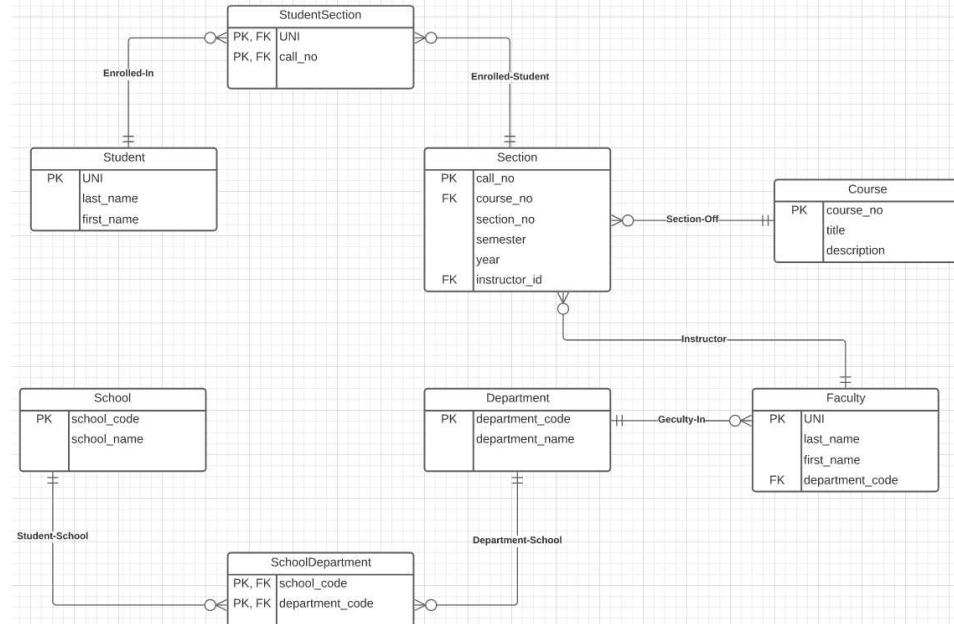
Primary Key means that the value occurs **at most once**.

| School Code | School Name                                             |
|-------------|---------------------------------------------------------|
| CC          | Columbia College                                        |
| SEAS        | Fu Foundation School of Engineering and Applied Science |
| GSAS        | Graduate School of Arts and Sciences                    |
| GS          | General Studies                                         |
| ....        | ....                                                    |

Foreign Key means that if a value occurs in **school\_id** for any row, there must be a row in School with **that key**.

| UNI   | Last name | First name | school_id |
|-------|-----------|------------|-----------|
| dff9  | Ferguson  | Donald     | CC        |
| js11  | Smith     | John       | GS        |
| jp9   | Public    | James      | CC        |
| bb101 | Baggins   | Bilbo      | CC        |
| ....  | ....      | ....       | ....      |

Student.school\_id references school.school\_code



The line notations mean:

- A student is related to EXACTLY ONE school.
- A School may be related to 0, 1 or many students.



# ER model -- Database Modeling

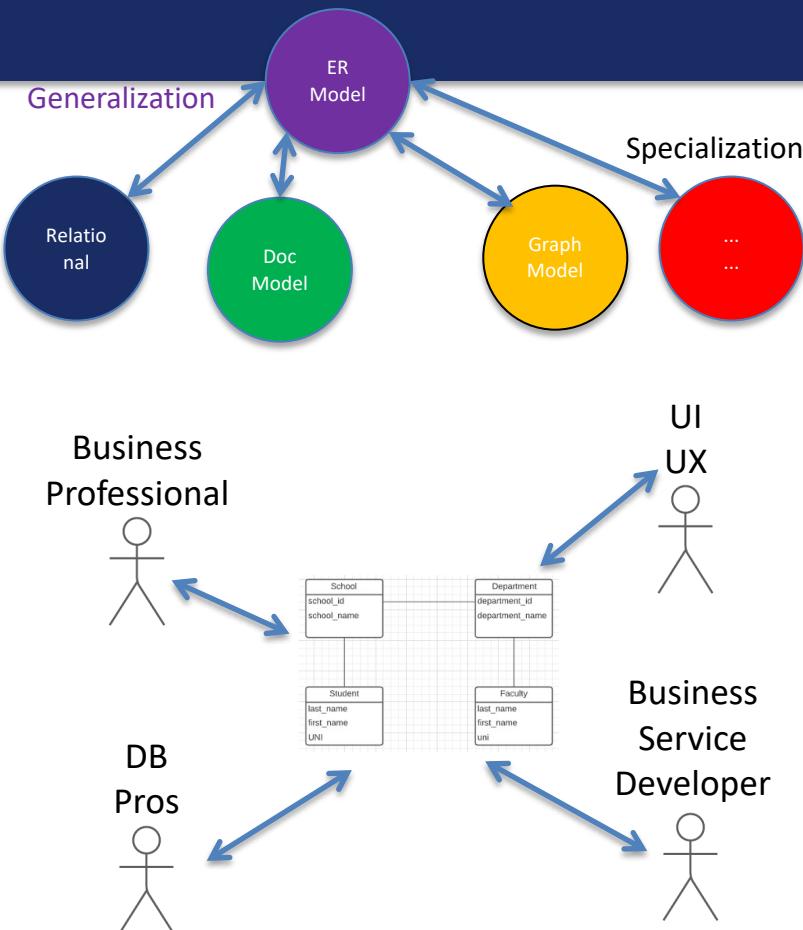
- The ER data mode was developed to facilitate database design by allowing specification of an **enterprise schema** that represents the overall logical structure of a database.
- The ER data model employs three basic concepts:
  - entity sets,
  - relationship sets,
  - attributes.
- The ER model also has an associated diagrammatic representation, the **ER diagram**, which can express the overall logical structure of a database graphically.

## DFF Comments:

- The book and slides do not do a great job of motivating the ER model or ER diagrams.
- Why do people and teams think about or use the ER model and modeling?

# ER Model and ER Modeling

- ER Model: Agility, Separation of Concerns
  - ER model is a generalization that most DB models implement in some form.
  - Using the ER model enables:
    - Thinking about and collaborating on design with getting bogged down in details.
    - Enable flexible choices about how to realize/implement data.
- ER Diagrams: Communication, Quality, Precision
  - With a little experience, everyone can understand an ER diagram.
  - Easier to discuss and collaborate on application's data than showing SQL table definitions, JSON, ... ...
  - People think visually. That is why we have whiteboards. ER diagrams are precise and unambiguous.
  - Guides you to think about relationships, keys, ... And prevents “re-dos” later in the process. It is easier to fix a diagram than a database schema.



# ER Modeling – Reasonably Good Summary

## Advantages of ER Model

**Conceptually it is very simple:** ER model is very simple because if we know relationship between entities and attributes, then we can easily draw an ER diagram.

**Better visual representation:** ER model is a diagrammatic representation of any logical structure of database. By seeing ER diagram, we can easily understand relationship among entities and relationship.

**Effective communication tool:** It is an effective communication tool for database designer.

**Highly integrated with relational model:** ER model can be easily converted into relational model by simply converting ER model into tables.

**Easy conversion to any data model:** ER model can be easily converted into another data model like hierarchical data model, network data model and so on.

## Disadvantages of ER Model

**Limited constraints and specification**

**Loss of information content:** Some information be lost or hidden in ER model

**Limited relationship representation:** ER model represents limited relationship as compared to another data models like relational model etc.

**No representation of data manipulation:** It is difficult to show data manipulation in ER model.

**Popular for high level design:** ER model is very popular for designing high level design

**No industry standard for notation**

<https://pctechnicalpro.blogspot.com/2017/04/advantages-disadvantages-er-model-dbms.html>

### Note:

- If you get to use Google to help with take home exams, HW, etc.
- I get to use Google to help with slides.

*Apply to –*  
*Game of Thrones/IMDB*  
*University Data Model*

*~/Dropbox/00NewProjects/CU-Course-Example/game-of-thrones/data*

# Game of Thrones

- Bottom-Up Data Mapping:
  - “Nouns” usually map to Entity/Entity Set.
  - Nouns inside other nouns often map to:
    - Attribute
    - Relationship
  - Verbs often map to relationships.
  - Adjectives usually map to properties.
- We will start with a subset of the information:
  - Game of Thrones:
    - Episodes
    - Characters
  - IMDB:
    - names\_basics
    - title\_basics
- Entities Sets
  - Character
  - Season
  - Episode
  - Scene
  - Location, Sublocation
  - ... ...
- Relationships
  - Character – Scene
  - Character – Character (e.g. KilledBy)
  - Season – IMDB Title
  - Character – IMDB Name
  - ... ...

# Game of Thrones

- Bottom-Up Data Mapping:

- IMDB: <https://developer.imdb.com/non-commercial-datasets/>

- Do not download.
    - Despite being “tiny” compared to the real world.
    - The datasets are too big for most laptops.

- We have info

- Game of Thrones: <https://github.com/jeffreylancaster/game-of-thrones>

- IMDB datasets
    - names\_basics
    - title\_basics

- Entities Sets

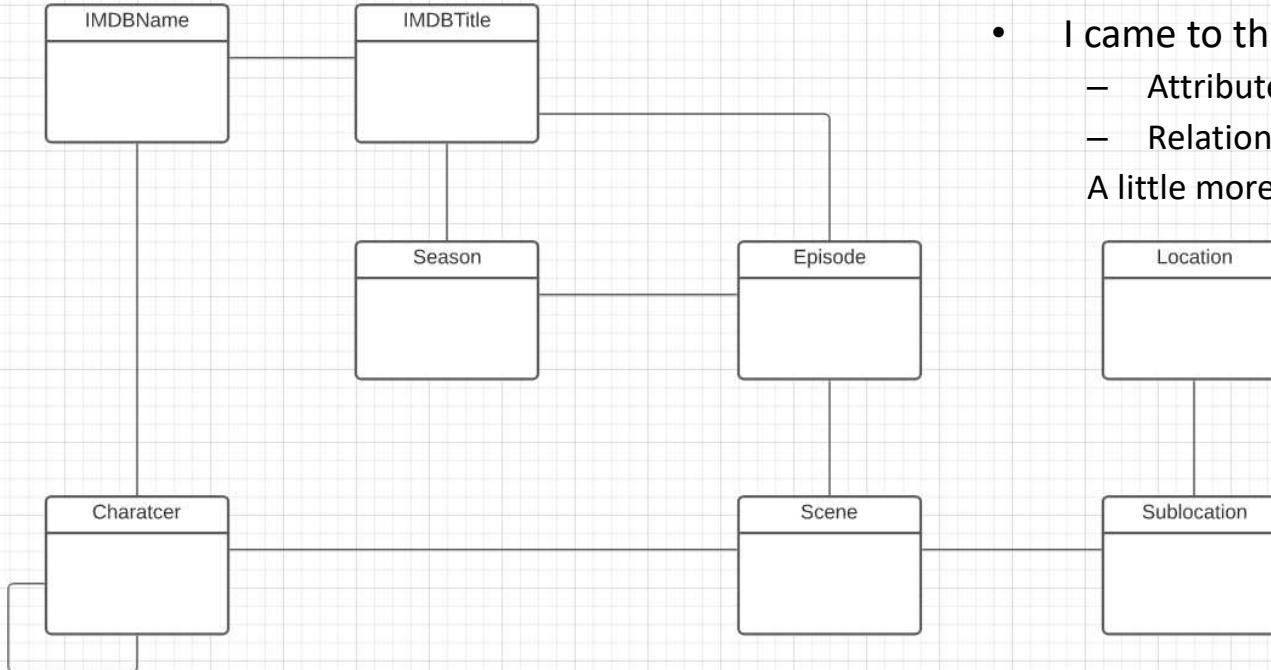
Character – IMDB Name

... ...

# Game of Thrones – Conceptual Model

Add shapes in Lucidchart ...

Add Entity Relationship Shapes

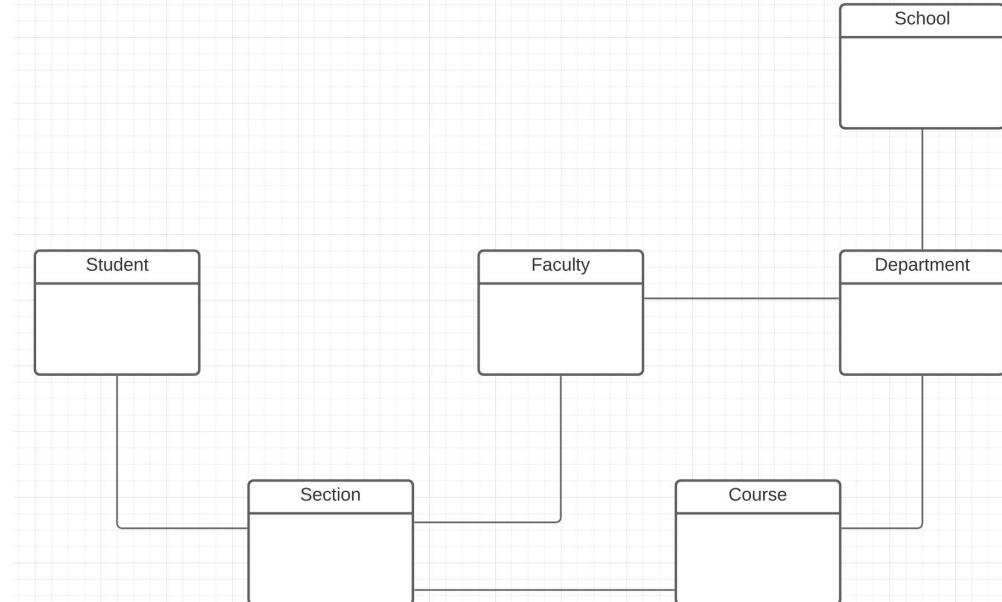


- With a little
  - Data exploration
  - Common sense
  - Judgment/experience
- I came to this conceptual model.
  - Attributes unspecified
  - Relationship required/cardinality unspecified.A little more exploration is needed.

# Columbia University – Conceptual Model

- A good first pass at a subset of entity types is:

- Student
- Faculty
- Class
- Section
- Department
- School



- And there are some obvious relationships

# Conceptual to Logical

<https://www.1keydata.com/datawarehousing/data-modeling-levels.html>

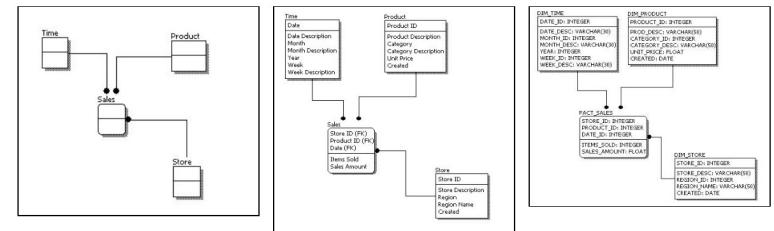
- The preceding material provided a conceptual model for a question on HW1
- HW 1 requests a *logical model*. We must add:
  - Attributes
  - Primary Keys
  - Foreign Keys
- HW 1 reinforces logical modeling concepts:
  - It is not trying to accurately model a university data model.
  - Do not worry about perfect semantics.
  - Just show you understand the concepts.
- To understand logical models, let's look at the Lahman's Baseball DB.

| Feature              | Conceptual | Logical | Physical |
|----------------------|------------|---------|----------|
| Entity Names         | ✓          | ✓       |          |
| Entity Relationships | ✓          | ✓       |          |
| Attributes           |            | ✓       |          |
| Primary Keys         |            | ✓       | ✓        |
| Foreign Keys         |            | ✓       | ✓        |
| Table Names          |            |         | ✓        |
| Column Names         |            |         | ✓        |
| Column Data Types    |            |         | ✓        |

Conceptual Model Design

Logical Model Design

Physical Model Design



<https://www.1keydata.com/datawarehousing/data-modeling-levels.html>

# Lahman's Baseball DB (You will also use)

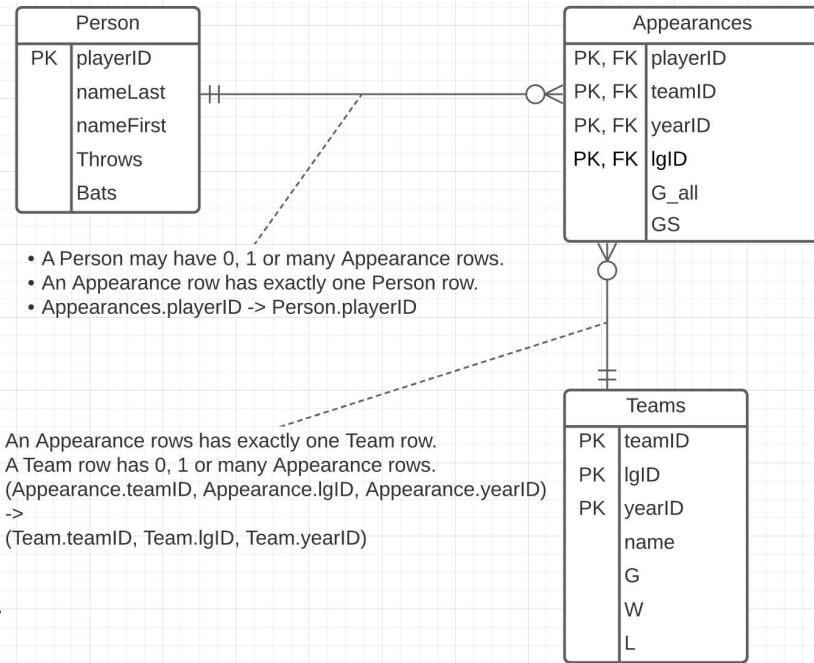
- Some core entity types are:
  - Person
  - Team
  - Appearances
- The relationship is simple:
  - Person – Appearance
  - Appearance – Team
- We will only do a subset of the attributes.

Switch to Notebook and run queries.

<https://www.seanlahman.com/baseball-archive/>

## Notes:

- The actual database's keys are different.
- I chose these keys for illustrative purposes.
- When we get deeper into the data and look at the logical model, we will need to make changes.



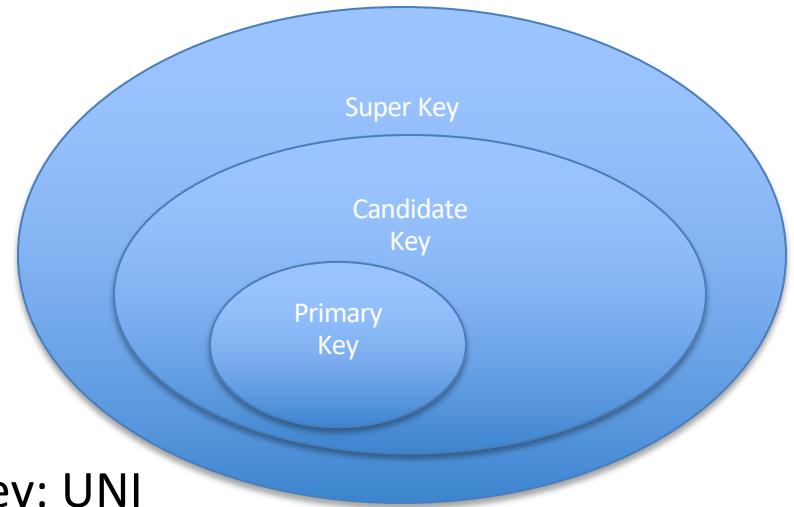
# Some Interesting Queries

- A super key “test:”
  - `select playerID, teamID, IgID, yearID, count(*) as count from appearances group by playerID, teamID, yearID order by count desc limit 10;`
- A candidate key “test:”
  - `select playerID, teamID, yearID, count(*) as count from appearances group by playerID, teamID, yearID order by count desc limit 10;`
- A foreign key “test:”
  - `select * from appearances where playerID not in (select playerID from people)`
  - `select * from appearances where not exists (select * from teams where appearances.teamID=teams.teamID and appearances.yearID=teams.yearID and appearances.IgID=teams.IgID)`

# Ideas

- Candidate Key
  - Unique
  - Minimal number of columns
  - (UNI)
- Super Key
  - (UNI, last\_name)
- Student
  - Student\_ID
  - UNI
  - Email Unique (never same one twice. But I may not know → NULL)

- (dff9, Ferguson, Donald, [dff@cs.columbia.edu](mailto:dff@cs.columbia.edu))
- (dff9, Ferguson, Donald, ...)
- (dff9, Ferguson, Douglas, ...)
  - (uni, last\_name) → Super Key
  - (uni) → Super Key
- Candidate is a minimal super key
  - (uni)
  - (email)
- Primary is arbitrarily chosen candidate key: UNI



*Used Lucidchart to show top-down  
modeling for a university*

*or*

*IMDB, GoT*

# Relation Model and Algebra

# *Schema Definition*

# Notation

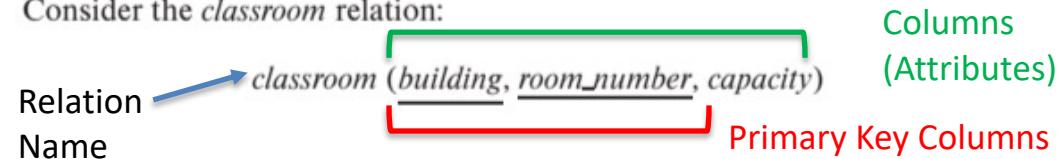
## Classroom relation

| building | room_number | capacity |
|----------|-------------|----------|
| Packard  | 101         | 500      |
| Painter  | 100         | 125      |
| Painter  | 514         | 10       |
| Taylor   | 3128        | 70       |
| Watson   | 100         | 30       |
| Watson   | 120         | 50       |

## classroom schema

It is customary to list the primary key attributes of a relation schema before the other attributes; for example, the *dept\_name* attribute of *department* is listed first, since it is the primary key. Primary key attributes are also underlined.

Consider the *classroom* relation:



- The primary key is a *composite key*. Neither column is a key (unique) by itself.
- Keys are statements about all possible, valid tuples and not just the ones in the relation.
  - Capacity is unique in this specific data, but clearly not unique for all possible data.
  - In this domain, there cannot be two classrooms with the same building and room number.
- Relation schema:
  - Underline indicates a primary key column. There is no standard way to indicate other types of key.
  - We will use **bold** to indicate foreign keys.
  - You will sometimes see things like *classroom*(building:string, room\_number:number, capacity:number)

*Some More Relational Algebra  
and  
The Dreaded RelaX Calculator*



# Relational Algebra

- A procedural language consisting of a set of operations that take one or two relations as input and produce a new relation as their result.
- Six basic operators
  - select:  $\sigma$
  - project:  $\Pi$
  - union:  $\cup$  (and intersection)
  - set difference:  $-$
  - Cartesian product:  $\times$
  - rename:  $\rho$
  - $\leftarrow$  assignment



$$Z = f(X, Y)$$

$$F: \mathbb{R}^2 \rightarrow \mathbb{R}$$



# Composition of Relational Operations

- The result of a relational-algebra operation is relation and therefore of relational-algebra operations can be composed together into a **relational-algebra expression**.
- Consider the query -- Find the names of all instructors in the Physics department.

$$\Pi_{name}(\sigma_{dept\_name = "Physics"}(instructor))$$

- Instead of giving the name of a relation as the argument of the projection operation, we give an expression that evaluates to a relation.

Why highlight composition before completing coverage of the operations?

- The next core relational operator most people consider is JOIN.
- The definition of JOIN relies on operation composition.



# Cartesian-Product Operation

- The Cartesian-product operation (denoted by  $\times$ ) allows us to combine information from any two relations.
- Example: the Cartesian product of the relations *instructor* and *teaches* is written as:  
$$\text{instructor} \times \text{teaches}$$
- We construct a tuple of the result out of each possible pair of tuples: one from the *instructor* relation and one from the *teaches* relation (see next slide)
- Since the instructor *ID* appears in both relations we distinguish between these attribute by attaching to the attribute the name of the relation from which the attribute originally came.
  - *instructor.ID*
  - *teaches.ID*

$$R \times R \rightarrow R \quad f(x,y) \rightarrow z$$



# The *instructor* X *teaches* table

- This only sort of makes sense. The result is:
  - Every possible combination of the form  
(instructor, teaches)
  - Even if the instructor is NOT the instructor  
in the teaches row.
- Examining in MySQL makes a little clearer.
  - Let's look in the lecture examples notebook.
  - Confusingly, in SQL  
Cartesian Product is **JOIN**

| Instructor.ID | name       | dept_name  | salary | teaches.ID | course_id | sec_id | semester | year |
|---------------|------------|------------|--------|------------|-----------|--------|----------|------|
| 10101         | Srinivasan | Comp. Sci. | 65000  | 10101      | CS-101    | 1      | Fall     | 2017 |
| 10101         | Srinivasan | Comp. Sci. | 65000  | 10101      | CS-315    | 1      | Spring   | 2018 |
| 10101         | Srinivasan | Comp. Sci. | 65000  | 10101      | CS-347    | 1      | Fall     | 2017 |
| 10101         | Srinivasan | Comp. Sci. | 65000  | 12121      | FIN-201   | 1      | Spring   | 2018 |
| 10101         | Srinivasan | Comp. Sci. | 65000  | 15151      | MU-199    | 1      | Spring   | 2018 |
| 10101         | Srinivasan | Comp. Sci. | 65000  | 22222      | PHY-101   | 1      | Fall     | 2017 |
| ...           | ...        | ...        | ...    | ...        | ...       | ...    | ...      | ...  |
| ...           | ...        | ...        | ...    | ...        | ...       | ...    | ...      | ...  |
| 12121         | Wu         | Finance    | 90000  | 10101      | CS-101    | 1      | Fall     | 2017 |
| 12121         | Wu         | Finance    | 90000  | 10101      | CS-315    | 1      | Spring   | 2018 |
| 12121         | Wu         | Finance    | 90000  | 10101      | CS-347    | 1      | Fall     | 2017 |
| 12121         | Wu         | Finance    | 90000  | 12121      | FIN-201   | 1      | Spring   | 2018 |
| 12121         | Wu         | Finance    | 90000  | 15151      | MU-199    | 1      | Spring   | 2018 |
| 12121         | Wu         | Finance    | 90000  | 22222      | PHY-101   | 1      | Fall     | 2017 |
| ...           | ...        | ...        | ...    | ...        | ...       | ...    | ...      | ...  |
| ...           | ...        | ...        | ...    | ...        | ...       | ...    | ...      | ...  |
| 15151         | Mozart     | Music      | 40000  | 10101      | CS-101    | 1      | Fall     | 2017 |
| 15151         | Mozart     | Music      | 40000  | 10101      | CS-315    | 1      | Spring   | 2018 |
| 15151         | Mozart     | Music      | 40000  | 10101      | CS-347    | 1      | Fall     | 2017 |
| 15151         | Mozart     | Music      | 40000  | 12121      | FIN-201   | 1      | Spring   | 2018 |
| 15151         | Mozart     | Music      | 40000  | 15151      | MU-199    | 1      | Spring   | 2018 |
| 15151         | Mozart     | Music      | 40000  | 22222      | PHY-101   | 1      | Fall     | 2017 |
| ...           | ...        | ...        | ...    | ...        | ...       | ...    | ...      | ...  |
| ...           | ...        | ...        | ...    | ...        | ...       | ...    | ...      | ...  |
| 22222         | Einstein   | Physics    | 95000  | 10101      | CS-101    | 1      | Fall     | 2017 |
| 22222         | Einstein   | Physics    | 95000  | 10101      | CS-315    | 1      | Spring   | 2018 |
| 22222         | Einstein   | Physics    | 95000  | 10101      | CS-347    | 1      | Fall     | 2017 |
| 22222         | Einstein   | Physics    | 95000  | 12121      | FIN-201   | 1      | Spring   | 2018 |
| 22222         | Einstein   | Physics    | 95000  | 15151      | MU-199    | 1      | Spring   | 2018 |
| 22222         | Einstein   | Physics    | 95000  | 22222      | PHY-101   | 1      | Fall     | 2017 |
| ...           | ...        | ...        | ...    | ...        | ...       | ...    | ...      | ...  |
| ...           | ...        | ...        | ...    | ...        | ...       | ...    | ...      | ...  |

# Simpler Example

|        |
|--------|
| T      |
| 4 rows |

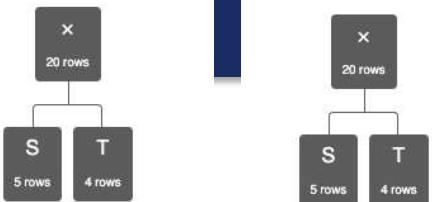
|        |
|--------|
| S      |
| 5 rows |

T

S

| T.b | T.d |
|-----|-----|
| 'a' | 100 |
| 'd' | 200 |
| 'f' | 400 |
| 'g' | 120 |

| S.b | S.d |
|-----|-----|
| 'a' | 100 |
| 'b' | 300 |
| 'c' | 400 |
| 'd' | 200 |
| 'e' | 150 |



$S \times T$

| S.b | S.d | T.b | T.d |
|-----|-----|-----|-----|
| 'a' | 100 | 'a' | 100 |
| 'a' | 100 | 'd' | 200 |
| 'a' | 100 | 'f' | 400 |
| 'a' | 100 | 'g' | 120 |
| 'b' | 300 | 'a' | 100 |
| 'b' | 300 | 'd' | 200 |
| 'b' | 300 | 'f' | 400 |
| 'b' | 300 | 'g' | 120 |
| 'c' | 400 | 'a' | 100 |
| 'c' | 400 | 'd' | 200 |
| 'c' | 400 | 'f' | 400 |
| 'c' | 400 | 'g' | 120 |
| 'e' | 150 | 'a' | 100 |
| 'e' | 150 | 'd' | 200 |
| 'e' | 150 | 'f' | 400 |
| 'e' | 150 | 'g' | 120 |

$S \times T$

| S.b | S.d | T.b | T.d |
|-----|-----|-----|-----|
| 'c' | 400 | 'f' | 400 |
| 'c' | 400 | 'g' | 120 |
| 'd' | 200 | 'a' | 100 |
| 'd' | 200 | 'd' | 200 |
| 'd' | 200 | 'f' | 400 |
| 'd' | 200 | 'g' | 120 |
| 'e' | 150 | 'a' | 100 |
| 'e' | 150 | 'd' | 200 |
| 'e' | 150 | 'f' | 400 |
| 'e' | 150 | 'g' | 120 |

- Assume we have two tables
  - S has two columns, 5 rows.
  - T has two columns, 4 rows.
- $S \times T$  has
  - 4 columns.
  - 20 rows.
- Cartesian product does not come up a lot in applications.
- There are cases in optimization in which:
  - You want to generate all possible combinations.
  - Score, rate, rank etc. to determine best choices.



# Join Operation

- The Cartesian-Product

*instructor X teaches*

associates every tuple of instructor with every tuple of teaches.

- Most of the resulting rows have information about instructors who did NOT teach a particular course.

- To get only those tuples of “*instructor X teaches*” that pertain to instructors and the courses that they taught, we write:

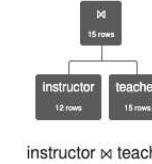
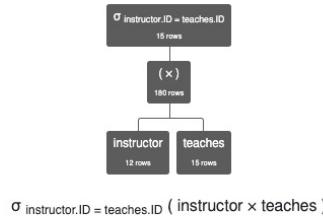
$\sigma_{instructor.id = teaches.id} (instructor \times teaches)$ )

- We get only those tuples of “*instructor X teaches*” that pertain to instructors and the courses that they taught.
- The result of this expression, shown in the next slide

A fundamental definition:

- $\sigma_{instructor.ID=teaches.ID} (instructor \times teaches) = instructor \bowtie teaches$
- $\bowtie$  is the JOIN operations.

# JOIN Definition



| instructor.ID | instructor.name | instructor.dept_name | instructor.salary | teaches.ID | teaches.course_id | teaches.sec_id | teaches.semester | teaches.year |
|---------------|-----------------|----------------------|-------------------|------------|-------------------|----------------|------------------|--------------|
| 10101         | 'Srinivasan'    | 'Comp. Sci.'         | 65000             | 10101      | 'CS-101'          | 1              | 'Fall'           | 2009         |
| 10101         | 'Srinivasan'    | 'Comp. Sci.'         | 65000             | 10101      | 'CS-315'          | 1              | 'Spring'         | 2010         |
| 10101         | 'Srinivasan'    | 'Comp. Sci.'         | 65000             | 10101      | 'CS-347'          | 1              | 'Fall'           | 2009         |
| 12121         | 'Wu'            | 'Finance'            | 90000             | 12121      | 'FIN-201'         | 1              | 'Spring'         | 2010         |
| 15151         | 'Mozart'        | 'Music'              | 40000             | 15151      | 'MU-199'          | 1              | 'Spring'         | 2010         |
| 22222         | 'Einstein'      | 'Physics'            | 95000             | 22222      | 'PHY-101'         | 1              | 'Fall'           | 2009         |
| 32343         | 'El Said'       | 'History'            | 60000             | 32343      | 'HIS-351'         | 1              | 'Spring'         | 2010         |
| 45565         | 'Katz'          | 'Comp. Sci.'         | 75000             | 45565      | 'CS-101'          | 1              | 'Spring'         | 2010         |
| 45565         | 'Katz'          | 'Comp. Sci.'         | 75000             | 45565      | 'CS-319'          | 1              | 'Spring'         | 2010         |
| 76766         | 'Crick'         | 'Biology'            | 72000             | 76766      | 'BIO-101'         | 1              | 'Summer'         | 2009         |

$\sigma_{instructor.ID=teaches.ID} (instructor \times teaches)$

| instructor.ID | instructor.name | instructor.dept_name | instructor.salary | teaches.course_id | teaches.sec_id | teaches.semester | teaches.year |
|---------------|-----------------|----------------------|-------------------|-------------------|----------------|------------------|--------------|
| 10101         | 'Srinivasan'    | 'Comp. Sci.'         | 65000             | 'CS-101'          | 1              | 'Fall'           | 2009         |
| 10101         | 'Srinivasan'    | 'Comp. Sci.'         | 65000             | 'CS-315'          | 1              | 'Spring'         | 2010         |
| 10101         | 'Srinivasan'    | 'Comp. Sci.'         | 65000             | 'CS-347'          | 1              | 'Fall'           | 2009         |
| 12121         | 'Wu'            | 'Finance'            | 90000             | 'FIN-201'         | 1              | 'Spring'         | 2010         |
| 15151         | 'Mozart'        | 'Music'              | 40000             | 'MU-199'          | 1              | 'Spring'         | 2010         |
| 22222         | 'Einstein'      | 'Physics'            | 95000             | 'PHY-101'         | 1              | 'Fall'           | 2009         |
| 32343         | 'El Said'       | 'History'            | 60000             | 'HIS-351'         | 1              | 'Spring'         | 2010         |
| 45565         | 'Katz'          | 'Comp. Sci.'         | 75000             | 'CS-101'          | 1              | 'Spring'         | 2010         |
| 45565         | 'Katz'          | 'Comp. Sci.'         | 75000             | 'CS-319'          | 1              | 'Spring'         | 2010         |
| 76766         | 'Crick'         | 'Biology'            | 72000             | 'BIO-101'         | 1              | 'Summer'         | 2009         |

$instructor \bowtie teaches$

$instructor \bowtie_{instructor.ID > teaches.ID} teaches$



## Join Operation (Cont.)

- The **join** operation allows us to combine a select operation and a Cartesian-Product operation into a single operation.
- Consider relations  $r(R)$  and  $s(S)$
- Let “theta” be a predicate on attributes in the schema R “union” S. The join operation  $r \bowtie_{\theta} s$  is defined as follows:

$$r \bowtie_{\theta} s = \sigma_{\theta}(r \times s)$$

- Thus

$$\sigma_{instructor.id = teaches.id}(instructor \times teaches))$$

- Can equivalently be written as

*instructor*  $\bowtie$  *Instructor.id = teaches.id teaches.*



# Union Operation

- The union operation allows us to combine two relations
- Notation:  $r \cup s$
- For  $r \cup s$  to be valid.
  1.  $r, s$  must have the **same arity** (same number of attributes)
  2. The attribute domains must be **compatible** (example: 2<sup>nd</sup> column of  $r$  deals with the same type of values as does the 2<sup>nd</sup> column of  $s$ )
- Example: to find all courses taught in the Fall 2017 semester, or in the Spring 2018 semester, or in both

$$\Pi_{course\_id} (\sigma_{semester="Fall"} \wedge year=2017(section)) \cup$$
$$\Pi_{course\_id} (\sigma_{semester="Spring"} \wedge year=2018(section))$$

( $a, b, c, d$ )

( $x, y$ )

*Project(a,*



# Union Operation (Cont.)

- Result of:

$$\Pi_{course\_id} (\sigma_{semester="Fall"} \wedge year=2017 (section)) \cup$$
$$\Pi_{course\_id} (\sigma_{semester="Spring"} \wedge year=2018 (section))$$

| course_id |
|-----------|
| CS-101    |
| CS-315    |
| CS-319    |
| CS-347    |
| FIN-201   |
| HIS-351   |
| MU-199    |
| PHY-101   |

Note: The preloaded dataset on the RelaX calculator is different from the most recent data referenced in the book. It is from a previous edition.



# Set-Intersection Operation

- The set-intersection operation allows us to find tuples that are in both the input relations.
- Notation:  $r \cap s$
- Assume:
  - $r, s$  have the *same arity*
  - attributes of  $r$  and  $s$  are compatible
- Example: Find the set of all courses taught in both the Fall 2017 and the Spring 2018 semesters.

$$\begin{aligned}\prod_{course\_id} (\sigma_{semester='Fall'} \wedge year=2017(section)) \cap \\ \prod_{course\_id} (\sigma_{semester='Spring'} \wedge year=2018(section))\end{aligned}$$

- Result

| course_id |
|-----------|
| CS-101    |



# Set Difference Operation

- The set-difference operation allows us to find tuples that are in one relation but are not in another.
- Notation  $r - s$
- Set differences must be taken between **compatible** relations.
  - $r$  and  $s$  must have the **same** arity
  - attribute domains of  $r$  and  $s$  must be compatible
- Example: to find all courses taught in the Fall 2017 semester, but not in the Spring 2018 semester

$$\Pi_{course\_id} (\sigma_{semester="Fall"} \wedge year=2017 (section)) -$$
$$\Pi_{course\_id} (\sigma_{semester="Spring"} \wedge year=2018 (section))$$

| <i>course_id</i> |
|------------------|
| CS-347           |
| PHY-101          |

# Same “arity”

Select DB (W4111 SimpleUnion) ▾

students

- `id` string
- `first_name` string
- `last_name` string
- `email` string
- `year` string

faculty

- `id` string
- `first_name` string
- `last_name` string
- `email` string
- `title` string
- `hire_date` string

- Same “arity”
  - Same number of columns.
  - Compatible types.
  - The  $i$ -th column in each table is from a compatible domain.
    - Student 5<sup>th</sup> column is “year.”
    - Faculty 5<sup>th</sup> column is “title”
    - Both are strings but combining them does not make sense.
- You can shape two incompatible tables using *project operations*. For example
  - $\pi \text{first\_name}, \text{last\_name}, \text{email} (\text{students})$   
   $\cap$   
     $\pi \text{first\_name}, \text{last\_name}, \text{email} (\text{faculty})$
  - $\pi \text{last\_name}, \text{email}, \text{title} \leftarrow \text{'Student'} (\text{students})$   
   $\cup$   
     $\pi \text{last\_name}, \text{email}, \text{title} (\text{faculty})$



# The Assignment Operation

- It is convenient at times to write a relational-algebra expression by assigning parts of it to temporary relation variables.
- The assignment operation is denoted by  $\leftarrow$  and works like assignment in a programming language.
- Example: Find all instructor in the “Physics” and Music department.

$$\text{Physics} \leftarrow \sigma_{\text{dept\_name} = \text{“Physics”}}(\text{instructor})$$
$$\text{Music} \leftarrow \sigma_{\text{dept\_name} = \text{“Music”}}(\text{instructor})$$
$$\text{Physics} \cup \text{Music}$$

- With the assignment operation, a query can be written as a sequential program consisting of a series of assignments followed by an expression whose value is displayed as the result of the query.

# What are all those other Symbols?

- $\tau$  order by
- $\gamma$  group by
- $\neg$  negation
- $\div$  set division
- $\bowtie$  natural join, theta-join
- $\bowtie_l$  left outer join
- $\bowtie_r$  right outer join
- $\bowtie_f$  full outer join
- $\bowtie_s$  left semi join
- $\bowtie_{rs}$  right semi join
- $\triangleright$  anti-join
- Some of these are pretty obscure
  - Division
  - Anti-Join
  - Left semi-join
  - Right semi-join
- Most SQL engines do not support them.
  - You can implement them using combinations of JOIN, SELECT, WHERE, ... ...
  - But, I cannot every remember using them in applications I have developed.
- Outer JOIN is very useful, but less common. We will cover.
- There are also some “patterns” or “terms”
  - Equijoin
  - Non-equi join
  - Natural join
  - Theta join
  - ... ...
- I may ask you to define these terms on some exams because they may be common internships/job interview questions.

# Some Terms

- “A NATURAL JOIN is a JOIN operation that creates an implicit join clause for you based on the common columns in the two tables being joined. Common columns are columns that have the same name in both tables.” (<https://docs.oracle.com/javadb/10.8.3.0/ref/rrefsqjnaturaljoin.html>)
- $\bowtie \rightarrow$  Natural Join in relational algebra.
- So, think about it ...
  - I showed you how to produce all possible pairs.
  - I showed you how to produce all naturally matching pairs.
  - Some simple set operations gives the anti-join.

# *SQL*

# *Core Concepts and Operations*



You can think of SQL being  
an extended, usable, useful version  
of the relational model.

## Chapter 3: Introduction to SQL

Database System Concepts, 7<sup>th</sup> Ed.

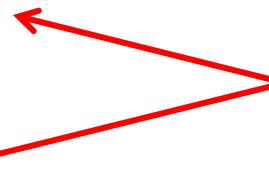
©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on re-use



# Outline

- Overview of The SQL Query Language
- SQL Data Definition
- Basic Query Structure of SQL Queries
- ~~Additional Basic Operations~~
- ~~Set Operations~~
- Null Values
- ~~Aggregate Functions~~
- ~~Nested Subqueries~~
- Modification of the Database

Will cover in next lecture  
and/or tutorial.





# History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
  - SQL-86
  - SQL-89
  - SQL-92
  - SQL:1999 (language name became Y2K compliant!)
  - SQL:2003
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
  - Not all examples here may work on your particular system.



# SQL Parts

- DML -- provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.
  - integrity – the DDL includes commands for specifying integrity constraints.
  - View definition -- The DDL includes commands for defining views.
  - Transaction control –includes commands for specifying the beginning and ending of transactions.
  - Embedded SQL and dynamic SQL -- define how SQL statements can be embedded within general-purpose programming languages.
  - Authorization – includes commands for specifying access rights to relations and views.
- 
- Data Definition Language (DDL)
  - Data Manipulation Language (DML)



# Data Definition Language

The SQL data-definition language (DDL) allows the specification of information about relations, including:

- The schema for each relation.
- The type of values associated with each attribute.
- The Integrity constraints
- The set of indices to be maintained for each relation.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.



# Domain Types in SQL

- **char(*n*)**. Fixed length character string, with user-specified length *n*.
- **varchar(*n*)**. Variable length character strings, with user-specified maximum length *n*.
- **int**. Integer (a finite subset of the integers that is machine-dependent).
- **smallint**. Small integer (a machine-dependent subset of the integer domain type).
- **numeric(*p,d*)**. Fixed point number, with user-specified precision of *p* digits, with *d* digits to the right of decimal point. (ex., **numeric(3,1)**, allows 44.5 to be stored exactly, but not 444.5 or 0.32)
- **real, double precision**. Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(*n*)**. Floating point number, with user-specified precision of at least *n* digits.
- More are covered in Chapter 4.



# Create Table Construct

- An SQL relation is defined using the **create table** command:

**create table** *r*

$(A_1 D_1, A_2 D_2, \dots, A_n D_n,$   
 $\text{(integrity-constraint}_1\text{)},$   
 $\dots,$   
 $\text{(integrity-constraint}_k\text{)})$

**Note to DFF:**  
Show performing tasks for  
new schema in dataGrip.

- r* is the name of the relation
- each  $A_i$  is an attribute name in the schema of relation *r*
- $D_i$  is the data type of values in the domain of attribute  $A_i$

- Example:

```
create table instructor (
 ID char(5),
 name varchar(20),
 dept_name varchar(20),
 salary numeric(8,2))
```



# Integrity Constraints in Create Table

- Types of integrity constraints
  - **primary key** ( $A_1, \dots, A_n$ )
  - **foreign key** ( $A_m, \dots, A_n$ ) **references**  $r$
  - **not null**
- SQL prevents any update to the database that violates an integrity constraint.
- Example:

```
create table instructor (
 ID char(5),
 name varchar(20) not null,
 dept_name varchar(20),
 salary numeric(8,2),
 primary key (ID),
 foreign key (dept_name) references department);
```



# And a Few More Relation Definitions

- ```
create table student (
    ID          varchar(5),
    name        varchar(20) not null,
    dept_name   varchar(20),
    tot_cred    numeric(3,0),
    primary key (ID),
    foreign key (dept_name) references department);
```

- ```
create table takes (
 ID varchar(5),
 course_id varchar(8),
 sec_id varchar(8),
 semester varchar(6),
 year numeric(4,0),
 grade varchar(2),
 primary key (ID, course_id, sec_id, semester, year) ,
 foreign key (ID) references student,
 foreign key (course_id, sec_id, semester, year) references section);
```



## And more still

- **create table** *course* (  
    *course\_id*      **varchar**(8),  
    *title*            **varchar**(50),  
    *dept\_name*     **varchar**(20),  
    *credits*        **numeric**(2,0),  
    **primary key** (*course\_id*),  
    **foreign key** (*dept\_name*) **references** *department*);



# Updates to tables

- **Insert**
  - `insert into instructor values ('10211', 'Smith', 'Biology', 66000);`
- **Delete**
  - Remove all tuples from the *student* relation
    - `delete from student`
- **Drop Table**
  - `drop table r`
- **Alter**
  - `alter table r add A D`
    - where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A*.
    - All existing tuples in the relation are assigned *null* as the value for the new attribute.
  - `alter table r drop A`
    - where *A* is the name of an attribute of relation *r*
    - Dropping of attributes not supported by many databases.



# Basic Query Structure

- A typical SQL query has the form:

```
select A_1, A_2, \dots, A_n
 from r_1, r_2, \dots, r_m
 where P
```

- $A_i$  represents an attribute
- $R_i$  represents a relation
- $P$  is a predicate.
- The result of an SQL query is a relation.

## Note:

- The SELECT ... FROM ... WHERE ... Combines two relational operators,  $\sigma$  and  $\Pi$ .
- Actually, it also combines other operators, e.g.  $\times$



# The select Clause

- The **select** clause lists the attributes desired in the result of a query
  - corresponds to the projection operation of the relational algebra
- Example: find the names of all instructors:

```
select name
 from instructor
```
- NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)
  - E.g.,  $Name \equiv NAME \equiv name$
  - Some people use upper case wherever we use bold font.



# The select Clause (Cont.)

- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, insert the keyword **distinct** after `select`.
- Find the department names of all instructors, and remove duplicates

```
select distinct dept_name
from instructor
```

- The keyword **all** specifies that duplicates should not be removed.

```
select all dept_name
from instructor
```



# The select Clause (Cont.)

- An asterisk in the select clause denotes “all attributes”

```
select *
from instructor
```

- An attribute can be a literal with no **from** clause

```
select '437'
```

- Results is a table with one column and a single row with value “437”
- Can give the column a name using:

```
select '437' as FOO
```

- An attribute can be a literal with **from** clause

```
select 'A'
from instructor
```

- Result is a table with one column and  $N$  rows (number of tuples in the *instructors* table), each row with value “A”



## The select Clause (Cont.)

- The **select** clause can contain arithmetic expressions involving the operation,  $+$ ,  $-$ ,  $*$ , and  $/$ , and operating on constants or attributes of tuples.
  - The query:

```
select ID, name, salary/12
 from instructor
```

would return a relation that is the same as the *instructor* relation, except that the value of the attribute *salary* is divided by 12.

- Can rename “*salary/12*” using the **as** clause:

```
select ID, name, salary/12 as monthly_salary
```



# The where Clause

- The **where** clause specifies conditions that the result must satisfy
  - Corresponds to the selection predicate of the relational algebra.
- To find all instructors in Comp. Sci. dept

```
select name
from instructor
where dept_name = 'Comp. Sci.'
```

- SQL allows the use of the logical connectives **and**, **or**, and **not**
- The operands of the logical connectives can be expressions involving the comparison operators <, <=, >, >=, =, and  $\neq$ .
- Comparisons can be applied to results of arithmetic expressions
- To find all instructors in Comp. Sci. dept with salary > 80000

```
select name
from instructor
where dept_name = 'Comp. Sci.' and salary > 80000
```



# The from Clause

- The **from** clause lists the relations involved in the query
  - Corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *instructor X teaches*

```
select *
 from instructor, teaches
```

  - generates every possible instructor – teaches pair, with all attributes from both relations.
  - For common attributes (e.g., *ID*), the attributes in the resulting table are renamed using the relation name (e.g., *instructor.ID*)
- Cartesian product not very useful directly, but useful combined with where-clause condition (selection operation in relational algebra).



# Examples

- Find the names of all instructors who have taught some course and the course\_id
  - **select** *name, course\_id*  
**from** *instructor , teaches*  
**where** *instructor.ID = teaches.ID*
- Find the names of all instructors in the Art department who have taught some course and the course\_id
  - **select** *name, course\_id*  
**from** *instructor , teaches*  
**where** *instructor.ID = teaches.ID and instructor.dept\_name = 'Art'*



# The Rename Operation

- The SQL allows renaming relations and attributes using the **as** clause:

*old-name as new-name*

- Find the names of all instructors who have a higher salary than some instructor in 'Comp. Sci'.

- **select distinct** *T.name*  
**from** *instructor as T, instructor as S*  
**where** *T.salary > S.salary and S.dept\_name = 'Comp. Sci.'*

- Keyword **as** is optional and may be omitted

*instructor as T*  $\equiv$  *instructor T*



# Null Values

- It is possible for tuples to have a null value, denoted by **null**, for some of their attributes
- **null** signifies an **unknown value** or that a **value does not exist**.
- The result of any arithmetic expression involving **null** is **null**
  - Example:  $5 + \text{null}$  returns **null**
- The predicate **is null** can be used to check for null values.
  - Example: Find all instructors whose salary is null.

```
select name
from instructor
where salary is null
```

- The predicate **is not null** succeeds if the value on which it is applied is not null.

## Note:

- **NULL is an extremely important concept.**
- **You will find it hard to understand for a while.**



# Null Values (Cont.)

- SQL treats as **unknown** the result of any comparison involving a null value (other than predicates **is null** and **is not null**).
  - Example:  $5 < \text{null}$  or  $\text{null} \neq \text{null}$  or  $\text{null} = \text{null}$
- The predicate in a **where** clause can involve Boolean operations (**and**, **or**, **not**); thus the definitions of the Boolean operations need to be extended to deal with the value **unknown**.
  - **and** :  $(\text{true and unknown}) = \text{unknown}$ ,  
 $(\text{false and unknown}) = \text{false}$ ,  
 $(\text{unknown and unknown}) = \text{unknown}$
  - **or**:  $(\text{unknown or true}) = \text{true}$ ,  
 $(\text{unknown or false}) = \text{unknown}$   
 $(\text{unknown or unknown}) = \text{unknown}$
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*



# Modification of the Database

- Deletion of tuples from a given relation.
- Insertion of new tuples into a given relation
- Updating of values in some tuples in a given relation



# Deletion

- Delete all instructors

**delete from** *instructor*

- Delete all instructors from the Finance department

**delete from** *instructor*  
**where** *dept\_name*= 'Finance';

- *Delete all tuples in the instructor relation for those instructors associated with a department located in the Watson building.*

**delete from** *instructor*  
**where** *dept\_name* **in** (**select** *dept\_name*  
**from** *department*  
**where** *building* = 'Watson');



## Deletion (Cont.)

- Delete all instructors whose salary is less than the average salary of instructors

```
delete from instructor
where salary < (select avg (salary)
 from instructor);
```

- Problem: as we delete tuples from *instructor*, the average salary changes
- Solution used in SQL:
  1. First, compute **avg** (*salary*) and find all tuples to delete
  2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)



# Insertion

- Add a new tuple to *course*

```
insert into course
values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

- or equivalently

```
insert into course (course_id, title, dept_name, credits)
values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

- Add a new tuple to *student* with *tot\_creds* set to null

```
insert into student
values ('3003', 'Green', 'Finance', null);
```



## Insertion (Cont.)

- Make each student in the Music department who has earned more than 144 credit hours an instructor in the Music department with a salary of \$18,000.

```
insert into instructor
 select ID, name, dept_name, 18000
 from student
 where dept_name = 'Music' and total_cred > 144;
```

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation.

Otherwise queries like

```
insert into table1 select * from table1
```

would cause problem



# Updates

- Give a 5% salary raise to all instructors

```
update instructor
 set salary = salary * 1.05
```

- Give a 5% salary raise to those instructors who earn less than 70000

```
update instructor
 set salary = salary * 1.05
 where salary < 70000;
```

- Give a 5% salary raise to instructors whose salary is less than average

```
update instructor
 set salary = salary * 1.05
 where salary < (select avg (salary)
 from instructor);
```



## Updates (Cont.)

- Increase salaries of instructors whose salary is over \$100,000 by 3%, and all others by a 5%
  - Write two **update** statements:

```
update instructor
 set salary = salary * 1.03
 where salary > 100000;
update instructor
 set salary = salary * 1.05
 where salary <= 100000;
```

- The order is important
- Can be done better using the **case** statement (next slide)

# *Aggregate Functions*



# Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value

**avg:** average value

**min:** minimum value

**max:** maximum value

**sum:** sum of values

**count:** number of values

Note: Some database implementations have additional aggregate functions.



# Aggregate Functions – Group By

- Find the average salary of instructors in each department
  - `select dept_name, avg (salary) as avg_salary  
from instructor  
group by dept_name;`

| <i>ID</i> | <i>name</i> | <i>dept_name</i> | <i>salary</i> |
|-----------|-------------|------------------|---------------|
| 76766     | Crick       | Biology          | 72000         |
| 45565     | Katz        | Comp. Sci.       | 75000         |
| 10101     | Srinivasan  | Comp. Sci.       | 65000         |
| 83821     | Brandt      | Comp. Sci.       | 92000         |
| 98345     | Kim         | Elec. Eng.       | 80000         |
| 12121     | Wu          | Finance          | 90000         |
| 76543     | Singh       | Finance          | 80000         |
| 32343     | El Said     | History          | 60000         |
| 58583     | Califieri   | History          | 62000         |
| 15151     | Mozart      | Music            | 40000         |
| 33456     | Gold        | Physics          | 87000         |
| 22222     | Einstein    | Physics          | 95000         |

| <i>dept_name</i> | <i>avg_salary</i> |
|------------------|-------------------|
| Biology          | 72000             |
| Comp. Sci.       | 77333             |
| Elec. Eng.       | 80000             |
| Finance          | 85000             |
| History          | 61000             |
| Music            | 40000             |
| Physics          | 91000             |

# Another View

## Employees

| DEPARTMENT_ID | SALARY |
|---------------|--------|
| 10            | 5500   |
| 20            | 15000  |
| 20            | 7000   |
| 30            | 12000  |
| 30            | 5100   |
| 30            | 4900   |
| 30            | 5800   |
| 30            | 5600   |
| 40            | 7500   |
| 40            | 8000   |
| 50            | 9000   |
| 50            | 8500   |
| 50            | 9500   |
| 50            | 8500   |
| 50            | 10500  |
| 50            | 10000  |
| 50            | 9500   |

Sum of Salary in Employees table for each department

| DEPARTMENT_ID | SUM(SALARY) |
|---------------|-------------|
| 10            | 5500        |
| 20            | 22000       |
| 30            | 33400       |
| 40            | 15500       |
| 50            | 65550       |

- GROUP BY column list
  - Forms partitions containing multiple rows.
  - All rows in a partition have the same values for the GROUP BY columns.
- The aggregate functions
  - Merge the non-group by attributes, which may differ from row to row.
  - Into a single value for each attribute.
- The result is one row per distinct set of GROUP BY values.
- There may be multiple non-GROUP BY COLUMNS, each with its own aggregate function.
- You can use HAVING in place of WHERE on the GROUP BY result.



# Aggregate Functions Examples

- Find the average salary of instructors in the Computer Science department
  - **select avg (salary)  
from instructor  
where dept\_name= 'Comp. Sci.';**
- Find the total number of instructors who teach a course in the Spring 2018 semester
  - **select count (distinct ID)  
from teaches  
where semester = 'Spring' and year = 2018;**
- Find the number of tuples in the *course* relation
  - **select count (\*)  
from course;**



# Aggregation (Cont.)

- Attributes in **select** clause outside of aggregate functions must appear in **group by** list

- /\* erroneous query \*/  
**select** *dept\_name, ID, avg (salary)*  
**from** *instructor*  
**group by** *dept\_name*;



# Aggregate Functions – Having Clause

- Find the names and average salaries of all departments whose average salary is greater than 42000

```
select dept_name, avg (salary) as avg_salary
from instructor
group by dept_name
having avg (salary) > 42000;
```

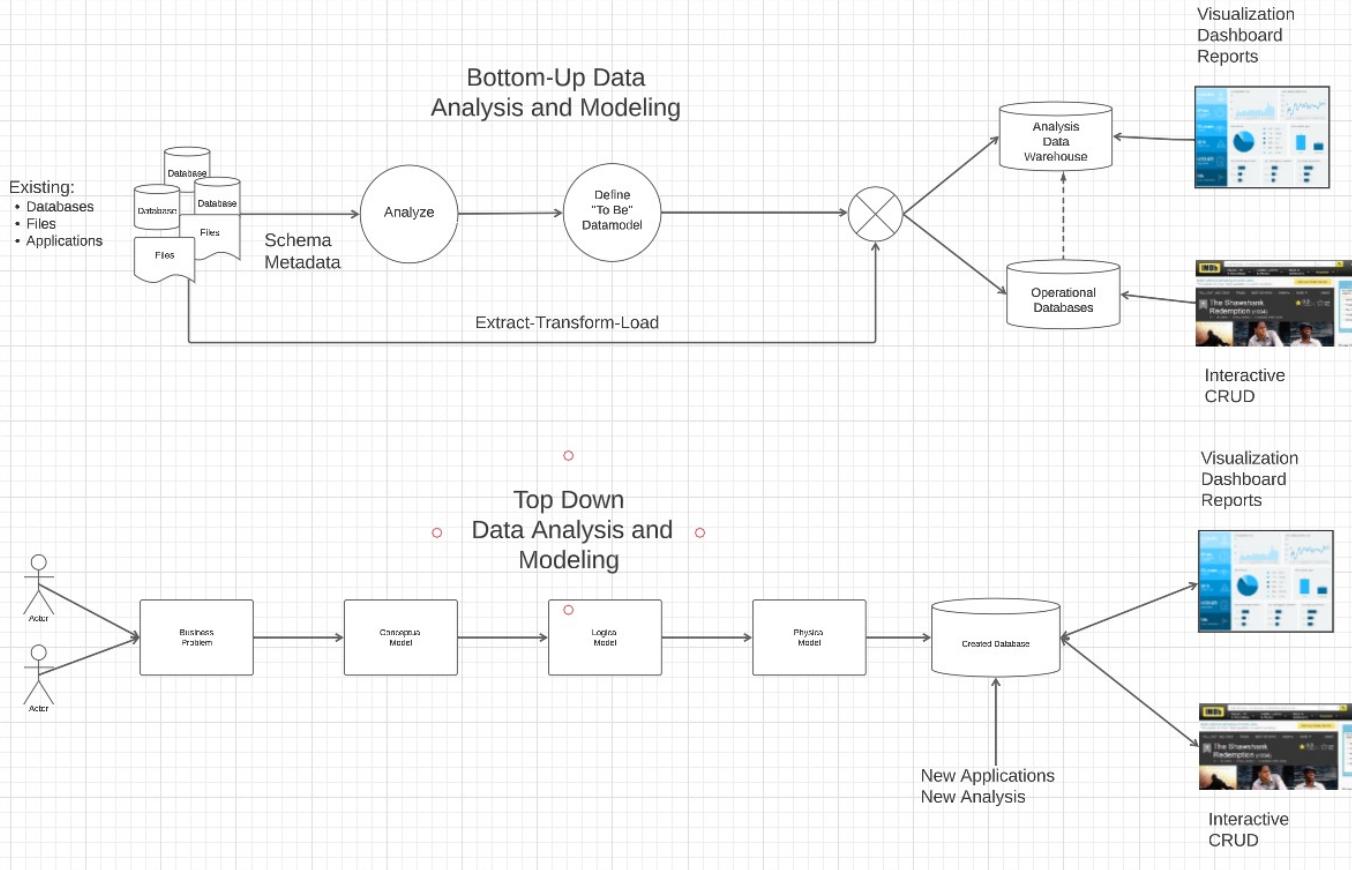
- Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

*Examples  
Web Application  
Data Engineering Notebook*

*The End for Today*

# *Motivating Problems*

# Top-Down and Bottom-Up Data Modeling



The most common is meet-in-the-middle:

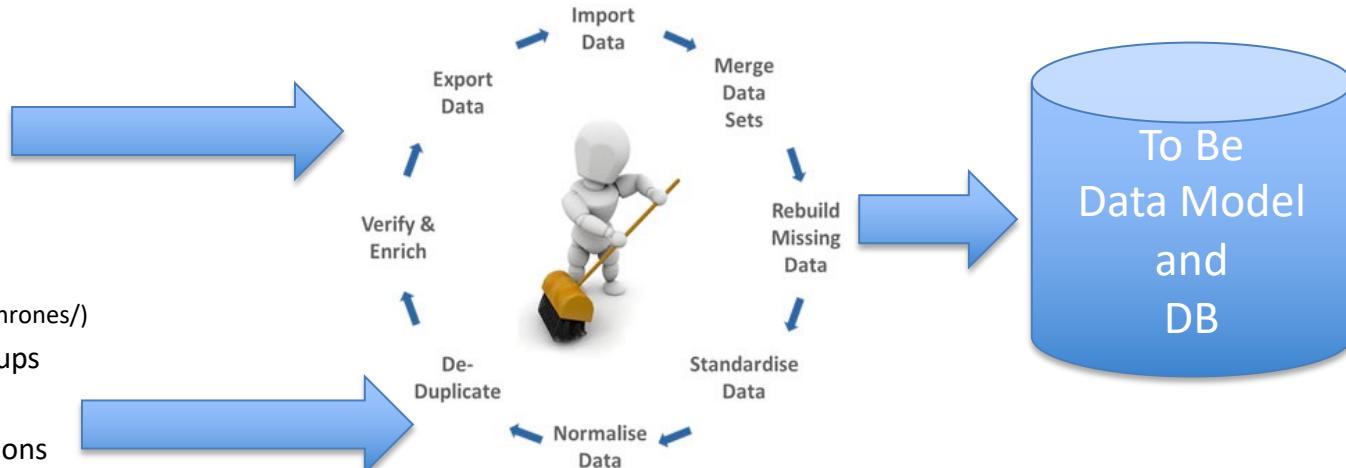
- Top-down defines the end state.
- Bottom-up defines how to get existing info there.
- Top-down development adds new apps and info.

# Bottom-Up: Game of Thrones and IMDB

Several files from IMDB:

- Titles Basics
- Titles AKAS
- Titles Episodes
- Titles Crews
- Title Principals
- Title Ratings
- Name Basics

Tab Separated Value



Game of Thrones:

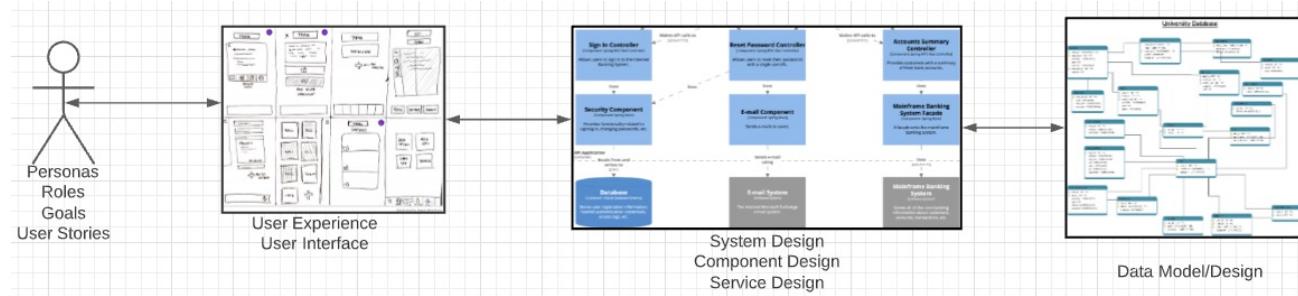
(<https://jeffreylancaster.github.io/game-of-thrones/>)

- Characters, Genders, Groups
- Episodes
- Locations, Opening Locations
- Words: Script, Bag, ... ....
- Geography
- ... ...

JSON

# Top-Down: University Management

- We must build a system that supports academic operations in a university.
- There are two major domains:
  - Interactive operations, e.g. register, choose class, assign grade, ... ...
  - Insight and reporting, e.g. enrollment trends, overloaded resources, ... ...
- We will design, develop, test and deploy the system iteratively and continuously.
- There are four core domains.



- The processes are iterative, with continuous extension and details.  
We will obviously focus on the data aspects in a database course.

- In this course,
- We focus on the data dimension.
- We will get some insight into the other dimensions.