W4111 – Introduction to Databases
Section 002, Spring 2021

Lecture 7: ER, Apps, SQL, Projects

COLUMBIA | ENGINEERING
The Fu Foundation School of Engineering and Applied Science

W4111 – Introduction to Databases
Section 002, Spring 2021

Lecture 7: ER, Apps, SQL, Projects

We will start in a couple of minutes.

Contents and Agenda

Contents

- Questions, comments, discussion.

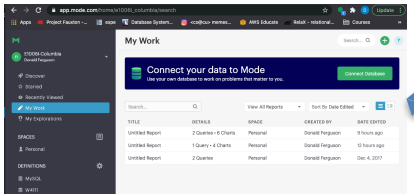
HW3 and HW4 – Incremental Project
My Example

My HW3 and HW4 Project Concept

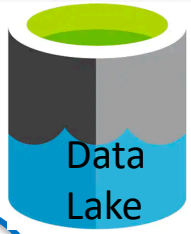
Pretend to Play Fantasy Baseball

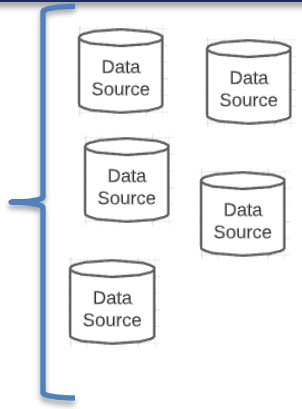
- My example projects for the two tracks will be a **simple** fantasy baseball solution.
  - "**Fantasy baseball** is a game in which people manage rosters of league baseball players, either online or in a physical location, using fictional fantasy baseball team names. The participants compete against one another using those players' real-life statistics to score points."
- My solution will have two subsystems:
  - *Analysis system*:
    - Read only data that enables people to select players based on performance, and to estimate the performance of their fantasy team.
    - Event based update system that changes analysis data based on new data.
  - *Operational system* that manages create, retrieve update, delete, etc. of their fantasy teams and leagues.
- INSERT, UPDATE, DELETE primarily apply to the operational system.

Target Application/Project(s) – Reminder (Lecture 1)



Data Insight Service





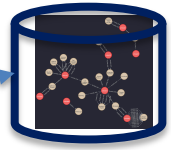
Web Application

Social Network Service

Core Data Service

Comment Discussion Service

DB Access Service



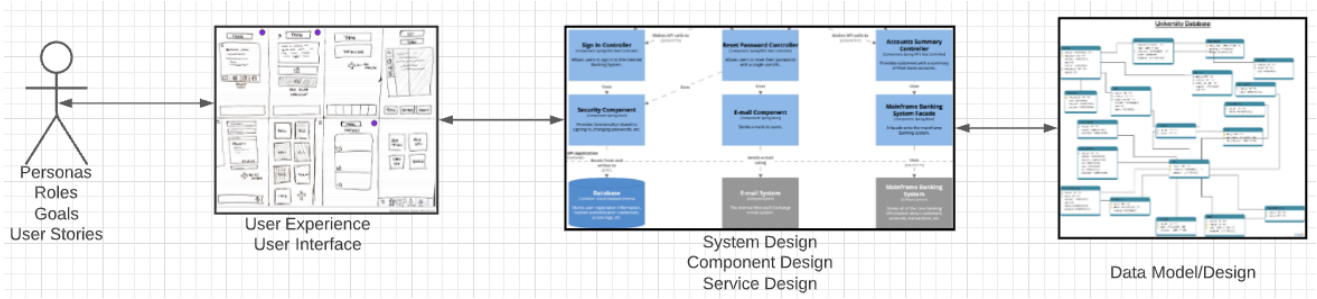
Target Application/Project – Reminder (Lecture 1)

- That diagram was pretty confusing.
- Basically, what it comes down to is that there will be major subsystems:
  1. Interactive web application for viewing, navigating and updating data.
     The updates have to preserve *semantic constraints* and correctness.
  2. A decision support warehouse/lake that allows us to explore data and get insights.
- Programming and non-programming tracks will get experience with both, but
  - Non-programming track focuses on data engineering needed to produce (2).
  - Programming track will focus on (1).
- The example that I will use is *fantasy baseball* because it covers both aspects, and there is a lot of interesting data available.
- You will have some freedom on the data models and scenarios you choose.

Operational System (Web Application)

Problem Statement – Modified (Lecture 1 Reminder)

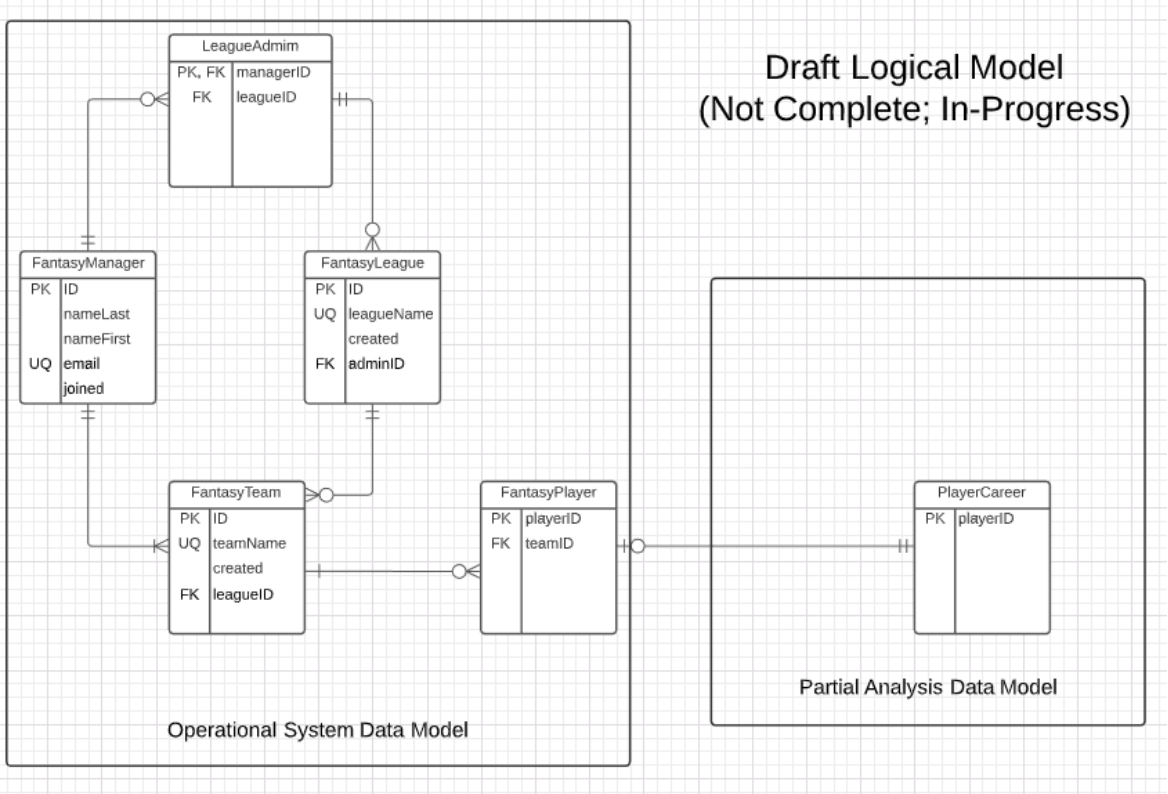
- We must build a system that supports ... ... operations in a ... ...
- We will design, develop, test and deploy the system iteratively and continuously.
- There are four core domains.



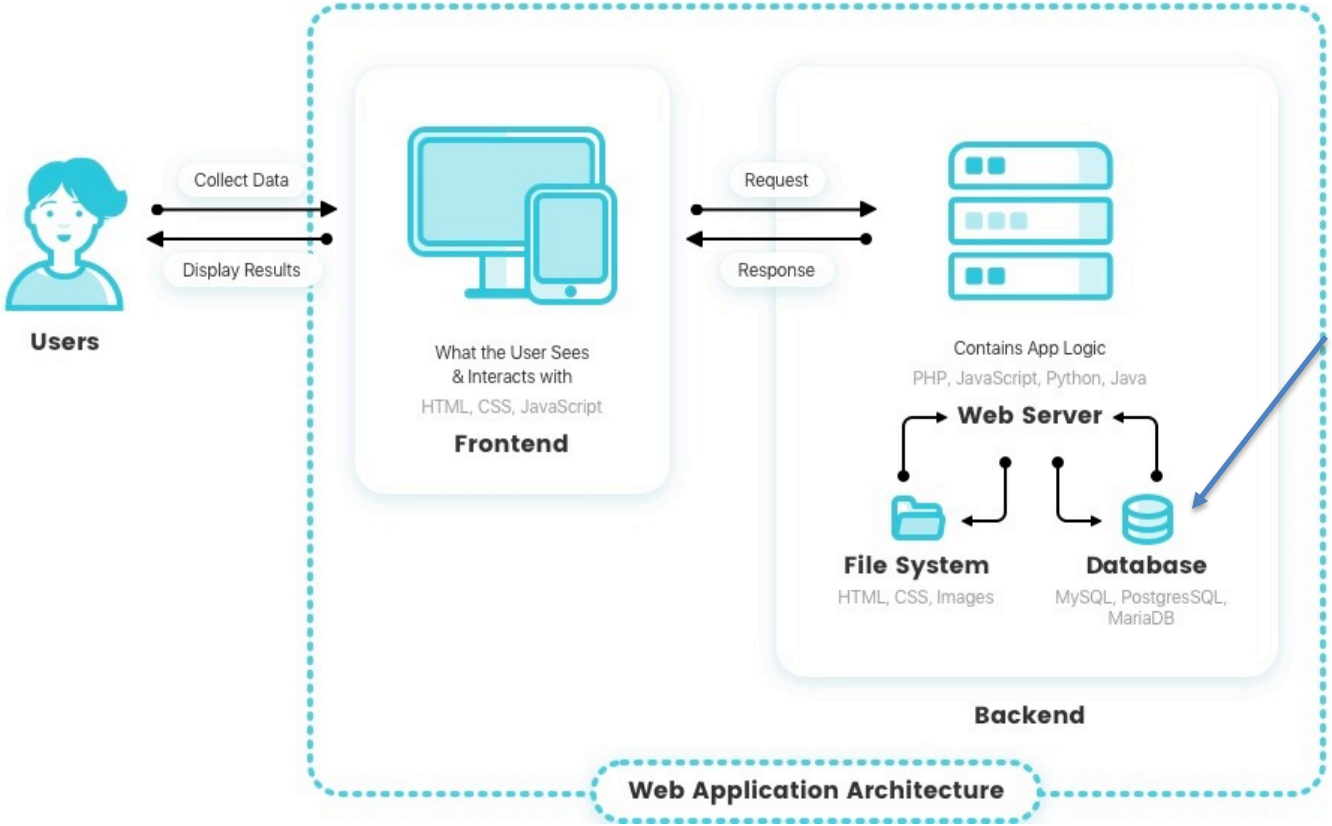
- In this course,
- We focus on the data dimension.
- We will get some insight into the other dimensions.

- The processes are iterative, with continuous extension and details.
- We will start implementing various *user stories.* Implementation requires:
  - Web UI
  - Paths
  - Data model and operations.

In-Progress Operational System Data Model



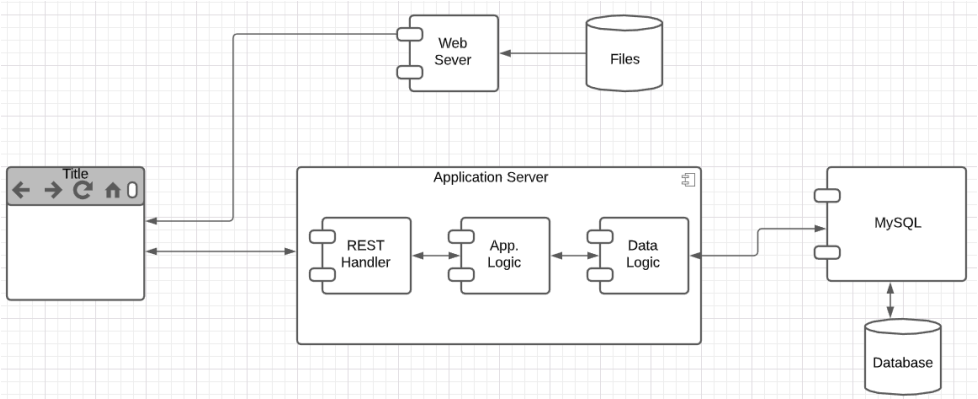
Web Application – Operational System



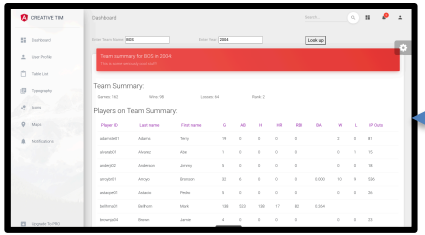
Lahman's Baseball DB
Fantasy Baseball DB

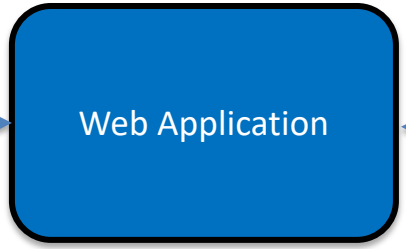
User Story

- "In software development and product management, a user story is an informal, natural language description of one or more features of a software system. User stories are often written from the perspective of an end user or user of a system. " (https://en.wikipedia.org/wiki/User_story)
- Example user stories that I need to implement for the operational system:
  - "As a fantasy team manager, I want to search for players based on career stats."
  - "As a fantasy team manager, I want to add a player to my fantasy team.
  - etc.
- I need to implement:
  - UI
  - Application logic
  - Database tables.



Simple Example





Two HTTP/REST calls:

- GET /api/team_summary?team_id=BOS&year_id=2004
- GET /api/teams?teamID=BOS&yearID=2004

Two SQL Queriess:

- SELECT to Teams table.
- SELECT to team_summary, which is a view. We cover views later.

- "As a baseball fan, I want to enter a teamID and yearID and see:
  - Team wins and losses.
  - Summary of player performance for players on the team and year."
- Built:
  - Dashboard page.
  - REST handlers and application logic.
  - Database view.

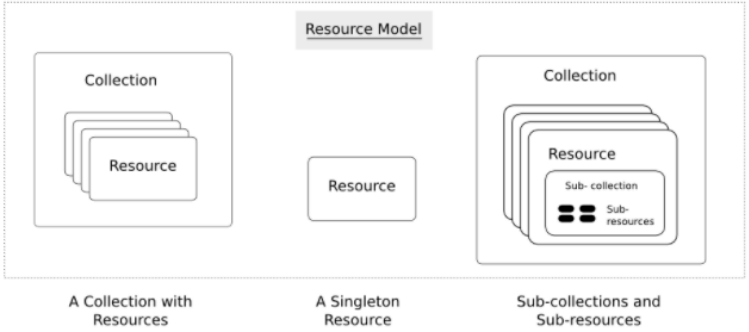
Tracks will build:

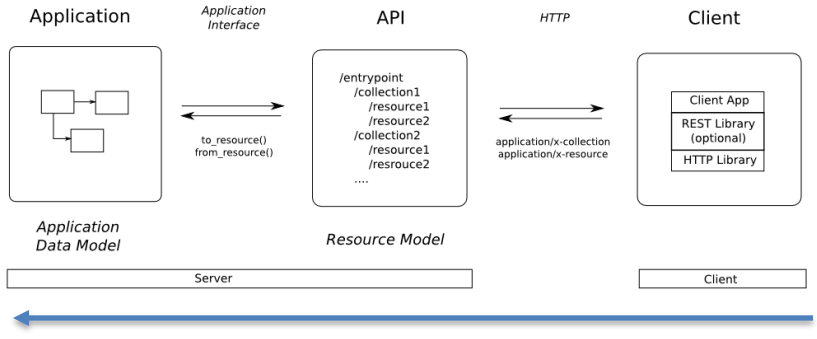
- Programming: Mix of CRUD and dashboard.
- Non-programming: Dashboards, Data transformation.

Will see more details as we move forward.

Simple Example – Fantasy Team Resource

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





Interaction flow is right-to-left in this diagram.

- Resources: (https://restful-api-design.readthedocs.io/en/latest/resources.html)
  - The fundamental concept in any RESTful API is the resource. A resource is an object with a type, associated data, relationships to other resources, and a set of methods that operate on it.
  - ... ... information that describes available resources types, their behavior, and their relationships the resource model of an API. The resource model can be viewed as the RESTful mapping of the application data model.
- APIs:
  - ... ... APIs expose functionality of an application or service that exists independently of the API. (DFF comment – the data)
  - Understanding enough of the important details of the application for which an API is to be created, so that an informed decision can be made as to what functionality needs to be exposed ... ...
  - Modeling this functionality in an API that addresses all use cases that come up in the real world ... ...

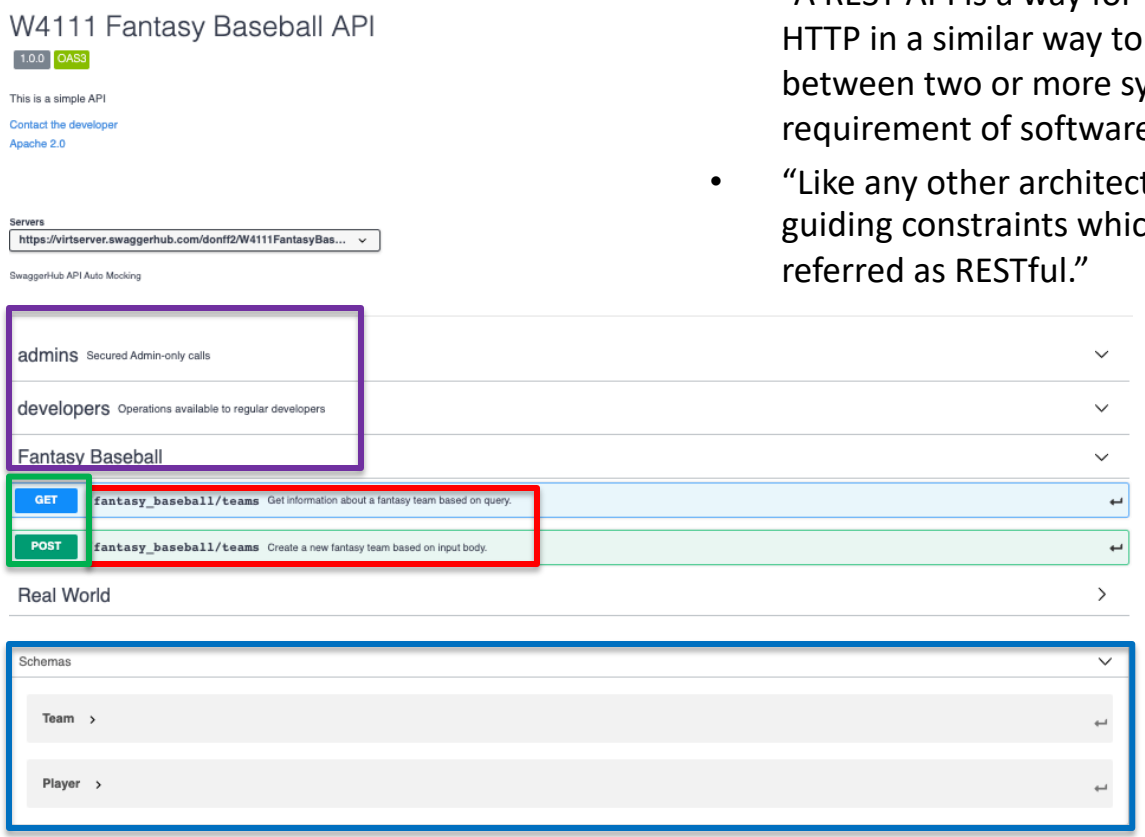
CRUD (https://en.wikipedia.org/wiki/Create,_read,_update_and_delete)

- Definitions:
  - "In computer programming, create, read, update, and delete[1] (CRUD) are the four basic functions of persistent storage."
  - "The acronym CRUD refers to all of the major functions that are implemented in relational database applications. Each letter in the acronym can map to a standard Structured Query Language (SQL) statement, Hypertext Transfer Protocol (HTTP) method (this is typically used to build RESTful APIs[5]) or Data Distribution Service (DDS) operation."

CRUD	SQL	HTTP	DDS
create	INSERT	PUT	write
read	SELECT	GET	read
update	UPDATE	PUT	write
delete	DELETE	DELETE	dispose

- Do not worry about Data Distribution Service.
- For our purposes HTTP – REST
- Entity Set:
  - Table in SQL
  - Collection Resource in REST.
- Entity:
  - Row in SQL
  - Resource in REST

REST API Definition



- "A REST API is a way for two computer systems to communicate over HTTP in a similar way to web browsers and servers. Sharing data between two or more systems has always been a fundamental requirement of software development.)
- "Like any other architectural style, REST also does have it's own 6 guiding constraints which must be satisfied if an interface needs to be referred as RESTful."

Open API Definition

- API Tags/Groupings
- A resource has
  - Paths
  - Methods
- Schema (Data Formats)
  - Sent on POST and PUT
  - Returned on GET

This material is just FYI and to help with understanding concepts, mapping to DB, ...

Demo and Next Steps

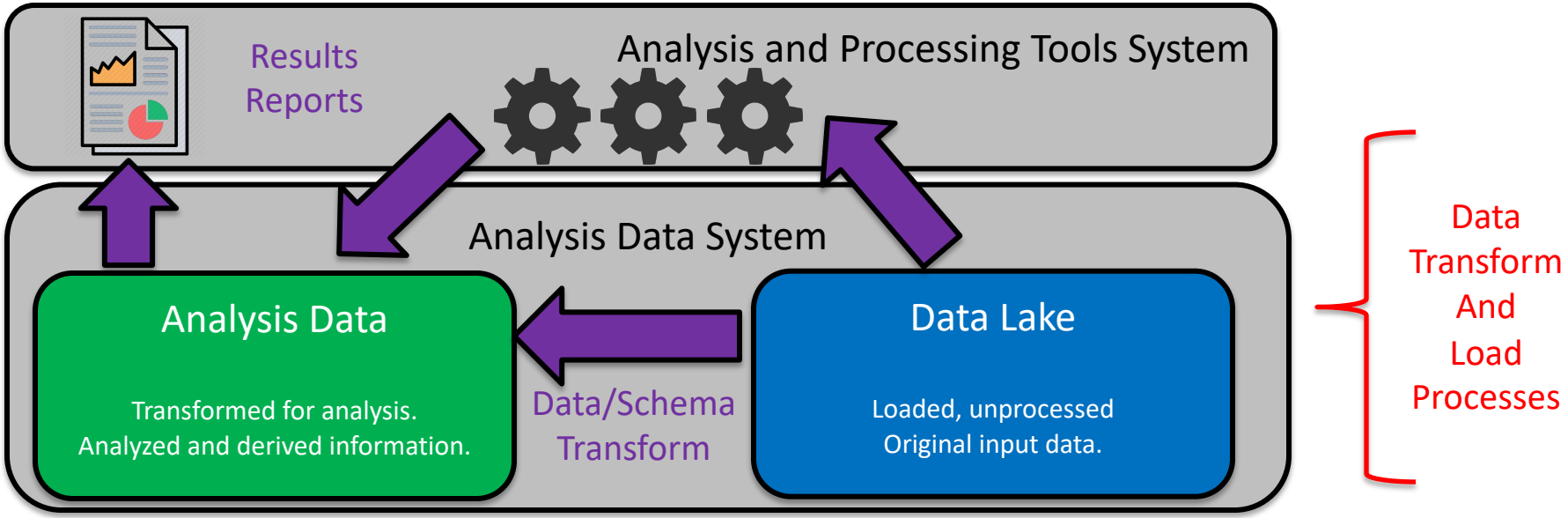
- API: https://app.swaggerhub.com/apis/donff2/W4111FantasyBaseball/1.0.0#/
- Starter API project:
  - App (Postman) http://127.0.0.1:5000/api/fantasy_team/BOS20011
  - Project: ~/Dropbox/Columbia/W4111_S21_New/W4111S21/Projects/FantasyBaseball

- Next Steps:
  - Will commit the template project, sample code and schedule orientation in recitation.
  - Will try to provide some UI, but this is not a UI class.

Analysis System

Analysis System – Fantasy Baseball Concept

- Focus is on the Analysis Data System (Primary Focus):
  - Data Lake is source data, imported and added to common database/model. (e.g. Lahman Baseball DB)
  - Analysis data is transformed data suitable for analysis, and analysis results. (e.g. Transformed Lahman's Data)
- Various analysis and processing tools use the data for insight, visualization, etc. (e.g. Jupyter, Pandas)



Data Lake to Analysis Data

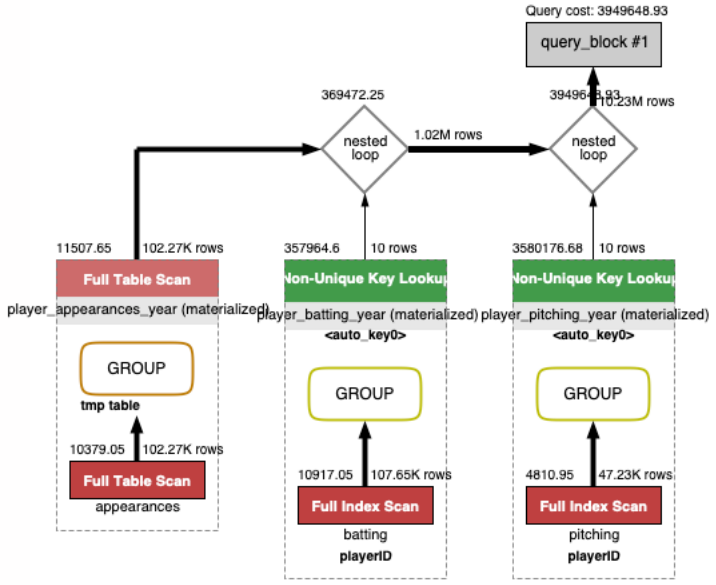
- This is a database class. So, the raw data to analysis data is a focus.
- Consider the following requirement.
  - I want to be able to predict a player's performance relative to salary based on history.
  - The first step is joining information from several tables, e.g.
    - Appearances.
    - Batting.
    - Pitching.
    - ~~Fielding~~. We are not trying to be super accurate about baseball. This is a DB class. We will ignore fielding.
    - Salary.
  - We then need to produce a single summary/prediction row. But, this is not as simple as just summing and averaging.
    - A year's numbers may be low because the player was injured and did not play many games.
    - The average is not meaningful. Over their career, player's get better and then worse. We want to use averages for historical players but trends for active players.
    - Salary data is missing. Average salary changed over time and we need to normalize.
    - **Absolute numbers do not matter. What matters is relative performance to other players**.

- **We will not do these now.**
- **I will do during other lectures and as I do my project.**

- Let's just do a very simple example. Career summary of annual performance relative to averages.

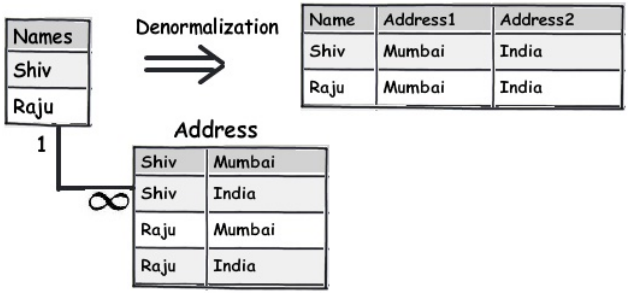
Task 1 – Player, Year Summary

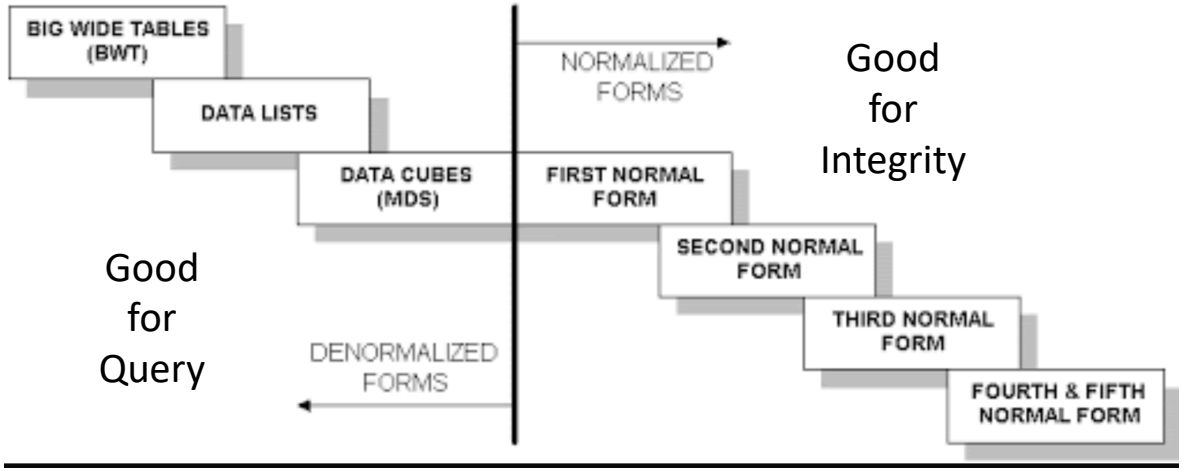
```
with player_basic
  as (
    select playerID, nameLast, nameFirst, bats, throws from people),
   player_appearances_year as
   (
     select
       playerid, yearid, sum(g_all) as app_g_all, sum(gs) as app_g_s, sum(g_defense) as app_g_defense,
       sum(g_p) as app_g_p, sum(g_c) as app_g_c, sum(g_1b) as app_g_1b, sum(g_2b) as app_g_2b,
         sum(g_3b) as app_g_3b, sum(g_ss), sum(g_lf) as app_g_lf, sum(g_cf) as app_g_cf,
           sum(g_rf) as app_g_rf
     from appearances group by playerid, yearid
   ),
   player_batting_year
  as
   (
     select playerid, yearid, sum(g) as bat_g, sum(g_batting) as g_batting,
          sum(ab) as bat_ab, sum(r) as bat_r, sum(h) as bat_h,
          sum(h-`2b`-`3b`-hr) as bat_1, sum(`2b`) as bat_2, sum(`3b`) as bat_3,
          sum(hr) as bat_hr, sum(rbi) as bat_rbi, sum(sh) as bat_sh, sum(sf) as bat_sf,
         sum(`2b`) as b2, sum(`3b`) as b3, sum(hr) as hr, sum(bb) as bb
    from batting group by playerid, yearid
  ),
  player_pitching_year
  as
    (
    select playerid, yearid, sum(w) as p_w, sum(l) as p_l, sum(g) as p_g, sum(IPOuts) as p_IPOuts,
      sum(SV) as p_SV, sum(h) as p_h, sum(hr) as p_hr, sum(bb) as p_bb, sum(er) as p_er
      from pitching group by playerid, yearid
       )
select * from
  (select * from player_appearances_year left join player_batting_year using(playerid, yearid)) as x
  left join
  player_pitching_year using(playerid, yearid)
```



- The core of the data lake are the raw tables.
- We clearly do not want to run this query every time we decide to examine and choose players.
- Task 1 adds a processed, "wide flat" table to the data lake for subsequent processing.

Wide Flat Tables





- Improve query performance by precomputing and saving:
  - JOINs
  - Aggregation
  - Derived/computed columns
- One of the primary strength of the relational model is maintaining "integrity" when applications create, update and delete data. This relies on:
  - The core capabilities of the relational model, e.g. constraints.
  - A well-design database (We will cover a formal definition – "normalization" in more detail later.
- Data models that are well designed for integrity are very bad for read only analysis queries.
  We will build and analyze wide flat tables as part of the analysis tasks in HW3, HW4 as projects.

External Information

- Analysis projects often include data from multiple sources and domains.
- Consider an example:
  - Building the best “fantasy baseball team” is an optimization problem.
  - Find the “best” players at the “lowest price (salary).”
- We have a few problems:
  - We must normalize old values to modern (constant) dollars.
  - There has been a general, upward trend in salaries. What matters is a player’s salary relative to the average salary for a year.
  - We have “missing” salaries:
    - No information before 1985.
    - Missing some entries in the range 1985-present.
- We will just do a hypothetical normalization right now.

Normalize the Value of the Dollar

- "A constant dollar is an adjusted value of currency used to compare dollar values from one period to another."
  (Investopedia)
- "To accurately compare income over time, users should adjust the summary measures (medians, means, etc.) for changes in cost of living (prices)."
  (US Census).
- This is not an MBA or economics class.
  We are just going to do something hypothetical and simple.
- "To use the CPI-U-RS to inflation adjust an income estimate from 1995 dollars to 2019 dollars, multiply the 1995 estimate by the CPI-U-RS from 2019 (376.5) divided by the CPI-U-RS from 1995 (225.3)"
  (US Census)

	CPI_US_I	Value
47	1993	215.5
48	1994	220
49	1995	225.3
50	1996	231.3
51	1997	236.3
52	1998	239.5
53	1999	244.6
54	2000	252.9
55	2001	260.1
56	2002	264.2
57	2003	270.2
58	2004	277.5
59	2005	286.9
60	2006	296.2
61	2007	304.6
62	2008	316.3
63	2009	315.2
64	2010	320.4
65	2011	330.5
66	2012	337.5
67	2013	342.5
68	2014	348.3
69	2015	348.9
70	2016	353.4
71	2017	361
72	2018	369.8
73	2019	376.5

- A 1995 salary of $100,000 is $ (376.5 / 225.3)*100000 = $167110.
- We will compute the inflation adjusted, average annual salary.

Hypothetical Normalization

```
with annual_average_salary as
  (
    select yearid, round(avg(salary),0) as avg_salary from
      lahmansbaseballdb.salaries group by yearid
  ),
normalized_averages as
      (
        select yearid, avg_salary, cpi.value  as salary_factor,
            round((avg_salary / cpi.value), 0) as normalized_avg_salary
        from annual_average_salary
            join cpi on yearid=cpi.CPI_US_I
      ),
averages_and_yoy as
  (
    select a.yearid, a.avg_salary, a.salary_factor, b.avg_salary as prior_avg_salary,
        b.salary_factor as prior_salary_factor
      from normalized_averages as a join normalized_averages as b
        on a.yearid=b.yearid+1
  )
select yearid, avg_salary salary_factor,
    round(avg_salary/prior_avg_salary,3) avg_salary_yoy,
    round(salary_factor/prior_salary_factor,3) as salary_factor_yoy
      from averages_and_yoy
order by yearid desc;
```

	yearid	salary_factor	avg_salary_yoy	salary_factor_yoy
1	2016	4396410	1.022	1.013
2	2015	4301276	1.081	1.002
3	2014	3980446	1.069	1.017
4	2013	3723344	1.077	1.015
5	2012	3458421	1.042	1.021
6	2011	3318838	1.012	1.032
7	2010	3278747	1	1.016
8	2009	3277647	1.045	0.997
9	2008	3136517	1.066	1.038
10	2007	2941436	1.038	1.028
11	2006	2834521	1.076	1.032
12	2005	2633831	1.057	1.034
13	2004	2491776	0.968	1.027

- I doubt very much that I got this correct.
- But you get the basic idea:
  - Process core data from the domain.
  - Bring in external information to adjust.
- Bring this together with the larger data, in this case to be able to talk about player performance relative to salary.

Hypothetical Player Choice

- Scenario:
  - I need a catcher for my team.
  - I want to find the “best 5 batting catchers by year.”
  - I measure batting performance by *slugging percentage.*
  - I want to normalize salary relative averages and CPI.
  - I do not want *absolute best.* I want best price performance, e.g. *(slugging percentage)/(normalized_salary)*

Switch to Notebook.

	yearid	playerid	nameLast	nameFirst	slg	relative_salary	slg_per_salary
1	2006	mccanbr01	McCann	Brian	0.5724	0.118	4.851
2	2001	pierzaj01	Pierzynski	A. J.	0.4409	0.092	4.793
3	2000	melusmi01	Meluskey	Mitch	0.4866	0.108	4.506
4	2000	estalbo02	Estalella	Bobby	0.4682	0.105	4.459
5	1993	piazzmi01	Piazza	Mike	0.5612	0.129	4.351

Optimal Price/Performance Catcher

```
with annual_batting_performance_salary as
     (
       select playerid,
            yearid,
            app_g_c,
            (bat_h / if(bat_ab = 0, NULL, bat_ab))                    as bat_avg,
            ((bat_h + bb) / (bat_ab + bb))                          as bat_obp,
            ((bat_1 + bat_2 * 2 + bat_3 * 3 + bat_hr * 4) / if(bat_ab = 0, NULL, bat_ab)) as slg,
            salary
       from player_performance_year
             join
           (select playerid, yearid, avg(salary) as salary
            from lahmansbaseballdb.salaries
            group by playerid, yearid) as a
           using
             (playerid, yearid)
       where bat_ab >= 100
     ),
normalized_salary_batting_performance as
  (
    select yearid, playerid,  bat_obp, slg, app_g_c,
        round(salary/normalized_salary_averages.salary_factor,3) as relative_salary from
         annual_batting_performance_salary join
          normalized_salary_averages using(yearid)
  )
select
  yearid, playerid, nameLast, nameFirst, slg, relative_salary, round(slg/relative_salary,3) as slg_per_salary
from normalized_salary_batting_performance join lahmansbaseballdb.people using(playerid)
  where app_g_c > 100
  order by slg_per_salary desc
limit 5;
```

- If I got that correct, it was a miracle.
- The purpose wasn't to win at Fantasy Baseball.
- Just giving a feel for the types of things that
  - Will happen in HW3, HW4 (Project) in
  - Analysis Subsystem
- The non-programming track will use tools:
  - Jupyter
  - SciKit
  - … …
  - To analyze.
- This is a database track. So, the programming track will also get some experience.
- Process for both tracks:
  - You can pick your own data and domain
  - Or follow along with me.
- There will be a set of criteria that you have to meet for each of HW4 and HW4, measure in:
  - Data model complexity.
  - Complexity of operations on the data.

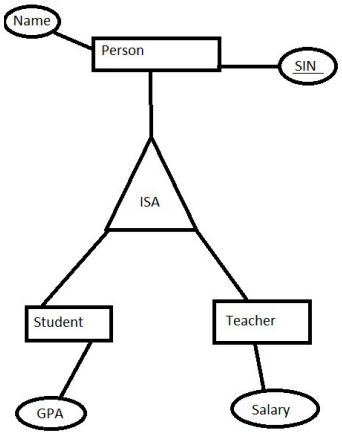
Complex Examples
Columbia Courses, Sections, Enrollments
(Part II – Students and Faculty)

IsA, Inheritance, Generalization, Specialization

Faculty and Students

- We modelled *course.*
- To *enroll* students in courses, we need ...
  - Sections
  - Faculty
  - Students
- Let's focus on the "people aspect." We have
  - People
  - A *Faculty* **is a** *People.*
  - A *Student* **is a** *People.*

ER Notation



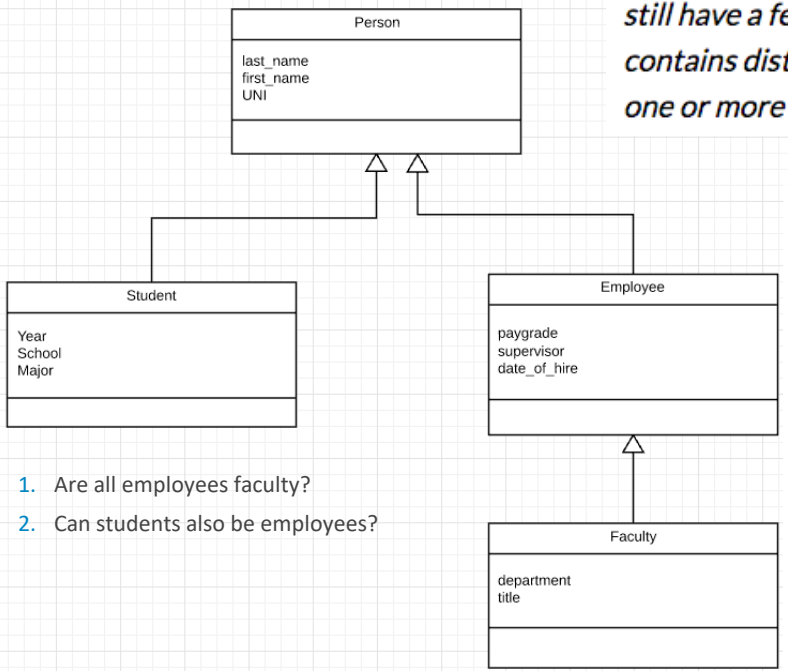
- "In object-oriented programming, inheritance is the mechanism of basing an object or class upon another object (prototype-based inheritance) or class (class-based inheritance), retaining similar implementation. Also defined as deriving new classes (sub classes) from existing ones such as super class or base class and then forming them into a hierarchy of classes."
- This is a new type of relationship in ER modeling.

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.

Inheritance, IsA, 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.

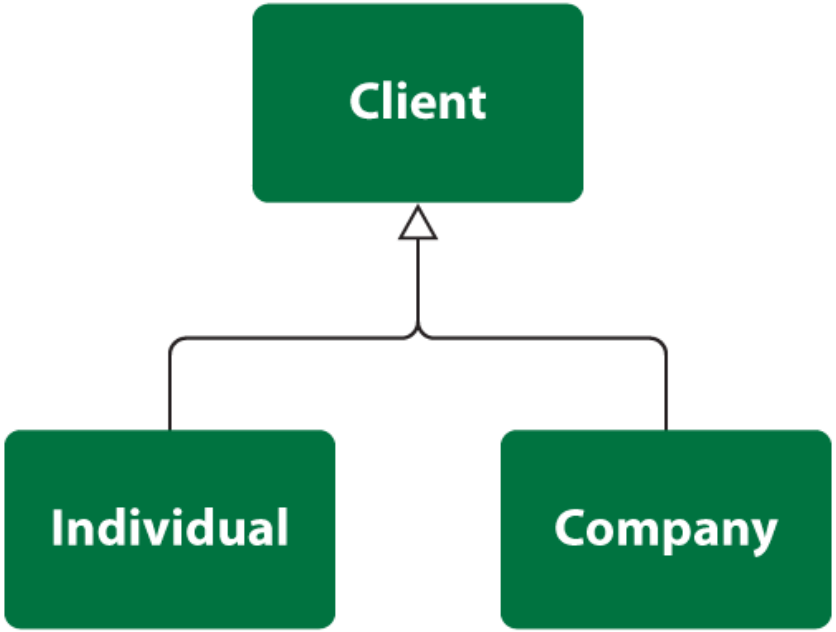
2 disjoint/overlapping

- 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.

http://www.vertabelo.com/blog/technical-articles/inheritance-in-a-relational-database

Simpler Example

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).

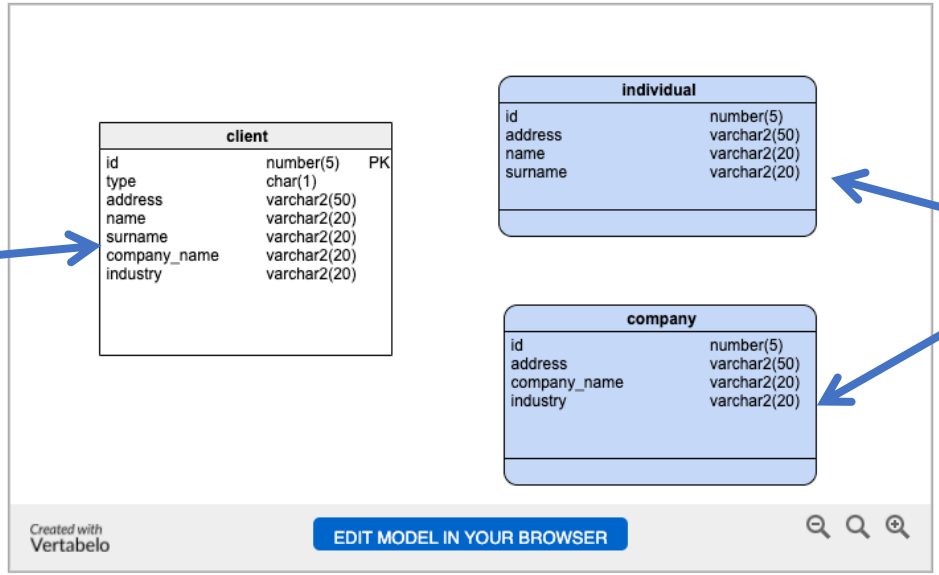


One Table Implementation

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`:



Tables

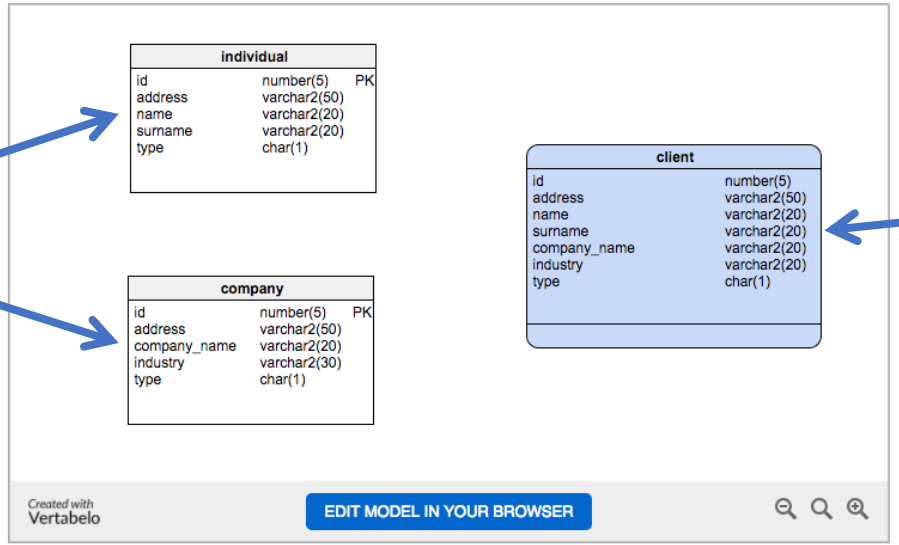
Views

Two Table Implementation

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`.



Tables

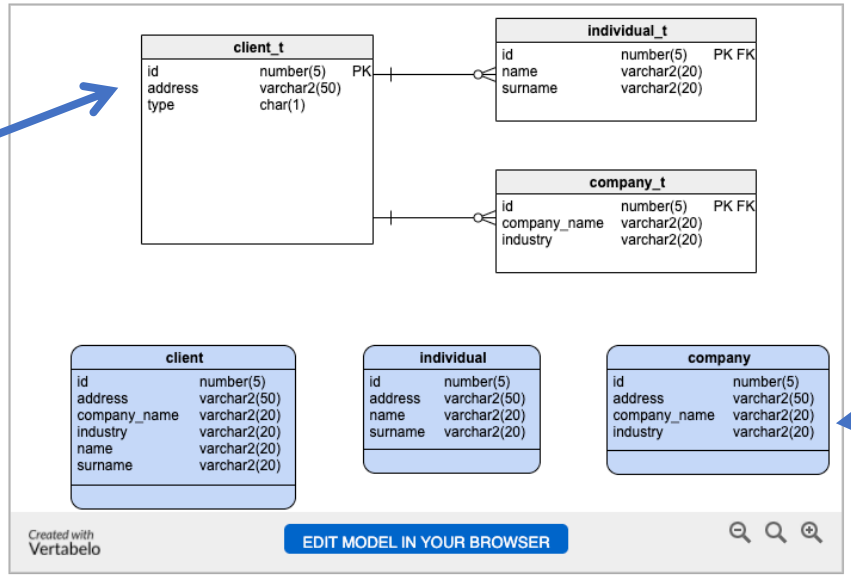
View

Two Table Implementation

Three-table implementation

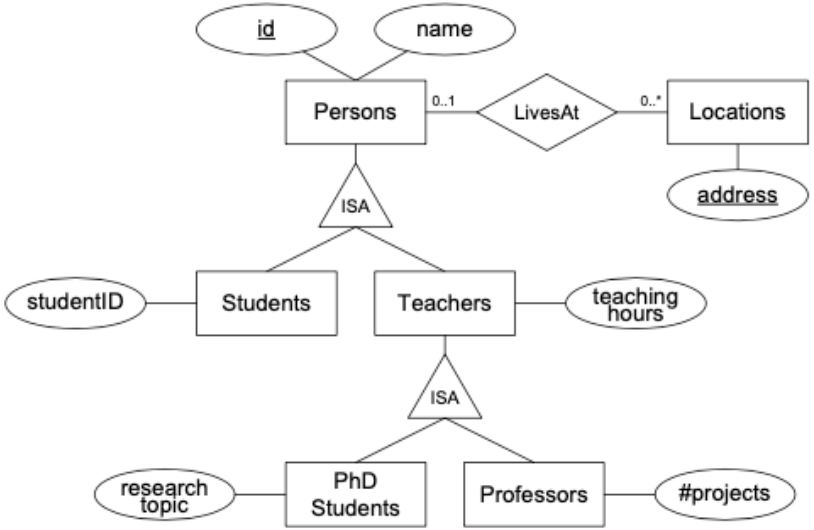
In a third solution we create a single table `client_t` for the parent table, containing common attributes for all subtypes, and tables for each subtype (`individual_t` and `company_t`) where the primary key in `client_t` (base table) determines foreign keys in dependent tables. There are three views: `client`, `individual` and `company`.

Tables

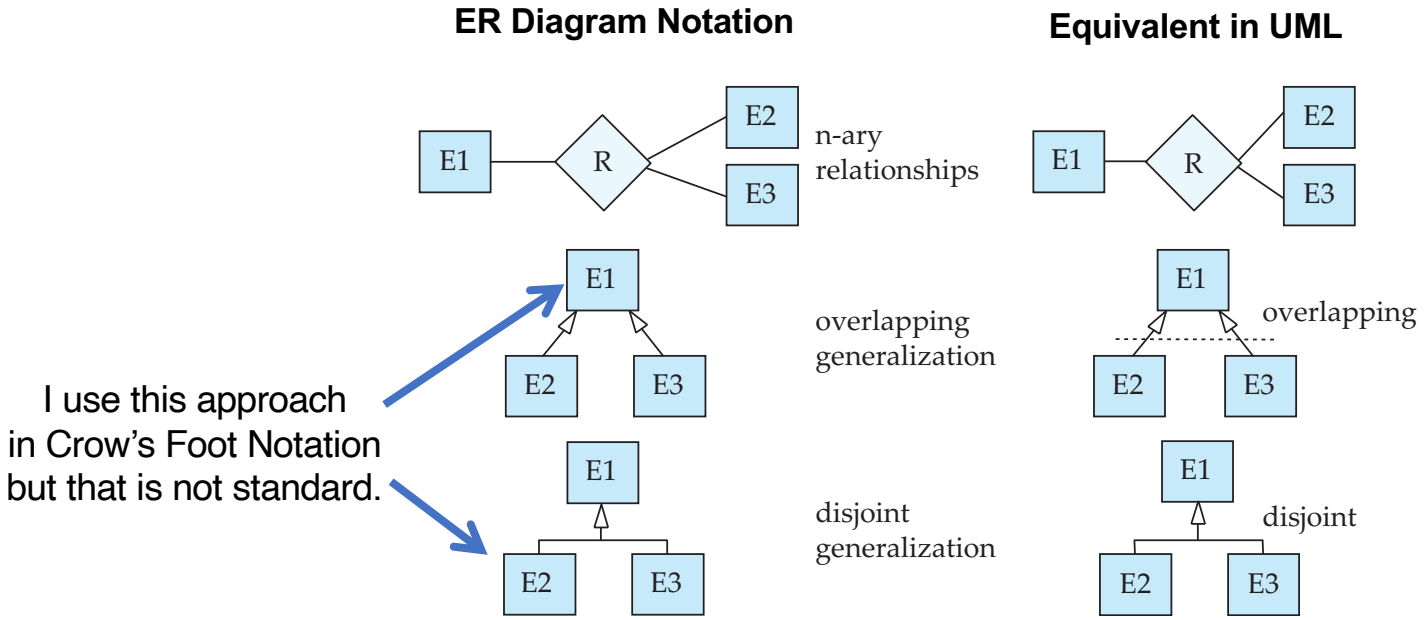


Views

ISA Relationship



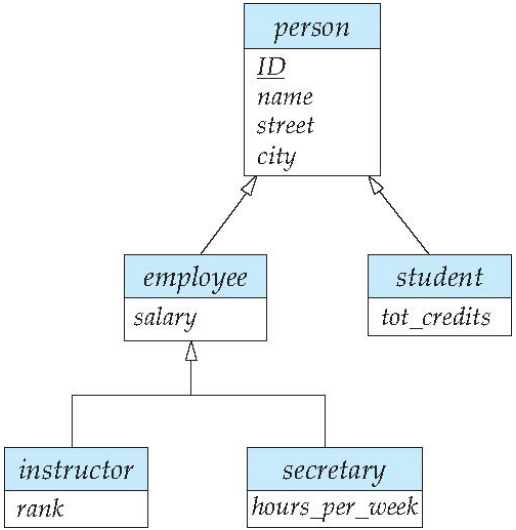
ER vs. UML Class Diagrams



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

Specialization Example

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



Generalization

- **A bottom-up design process** – combine a number of entity sets that share the same features into a higher-level entity set.
- Specialization and generalization are simple inversions of each other; they are represented in an E-R diagram in the same way.
- The terms specialization and generalization are used interchangeably.

Completeness constraint

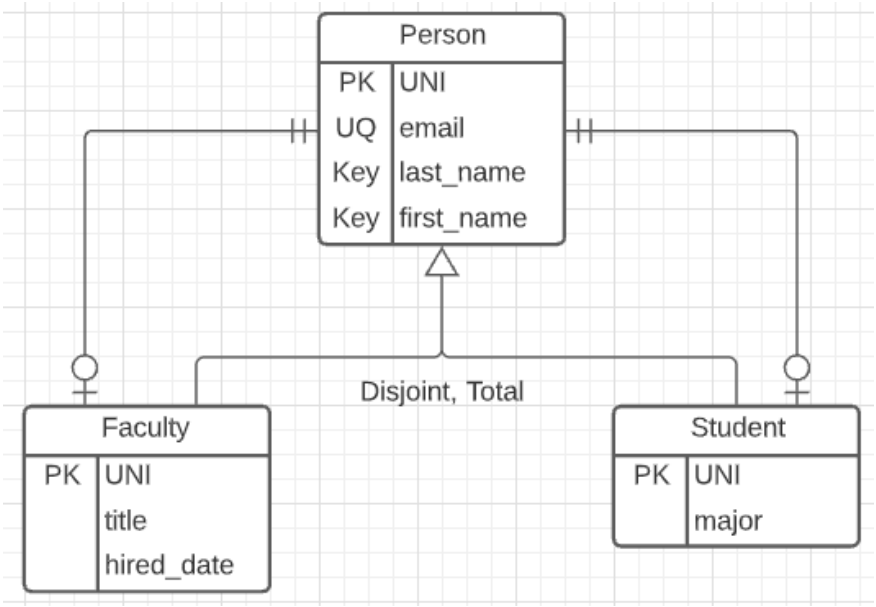
- **Completeness constraint** -- specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within a generalization.
  - **total**: an entity must belong to one of the lower-level entity sets
  - **partial**: an entity need not belong to one of the lower-level entity sets

Completeness constraint (Cont.)

- Partial generalization is the default.
- We can specify total generalization in an ER diagram by adding the keyword **total** in the diagram and drawing a dashed line from the keyword to the corresponding hollow arrow-head to which it applies (for a total generalization), or to the set of hollow arrow-heads to which it applies (for an overlapping generalization).
- The *student* generalization is total: All student entities must be either graduate or undergraduate. Because the higher-level entity set arrived at through generalization is generally composed of only those entities in the lower-level entity sets, the completeness constraint for a generalized higher-level entity set is usually total

Worked Example

Person, Faculty, Student



- Model
  - Complete:
    - Every entity is a Faculty or a Student.
    - There are no entities that are just Person.
  - Disjoint:
    - Either a Faculty or a Student
    - But not both.
  - 3-Table solution.
- Relationships:
  - Faculty-Person is one-to-one.
  - Student-Person is one-on-one.
  - Person is exactly 1.
  - Faculty or Student is 0-1.
- There is no easy way to handle the fact that one of Faculty or Student must exist.

The UNI

- The UNI creates some interesting challenges:
  - A given UNI is in two tables: Person, Faculty or Person, Student.
  - A given UNI can only be in the Faculty table or the Student table.
  - My algorithm for generating the UNI is:
    - Prefix: Concatenate
      - First two characters of first name.
      - First two characters of the last name
    - Suffix: Add 1 to the number of existing UNIs with the same prefix.
    - Concatenate the prefix and suffix.
  - Immutable: Cannot change.
- Implementing the semantics in applications accessing the data is error prone.
  - The more places you implement something, the more places you can mess up.
  - You have to find all the apps and change them if the logic changes.
  - So, we are going to implement using a function and trigger.

Table Definitions

```
CREATE TABLE `person_three` (
  `first_name` varchar(128) NOT NULL,
  `last_name` varchar(128) NOT NULL,
  `preferred_email` varchar(256) DEFAULT NULL,
  `uni` varchar(12) NOT NULL,
  `uni_email` varchar(256) GENERATED ALWAYS AS (concat(`uni`,_utf8mb4'@columbia.edu')) STORED,
  `preferred_name` varchar(128) DEFAULT NULL,
  PRIMARY KEY (`uni`),
  UNIQUE KEY `uni_email_UNIQUE` (`uni_email`),
  CONSTRAINT `person_three_chk_2` CHECK (((`preferred_email` is null) or
      ((`preferred_email` like _utf8mb4'%@%') and (not((`preferred_email` like _utf8mb4'%@%@%'))))))
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

CREATE TABLE `faculty_three` (
  `uni` varchar(12) NOT NULL,
  `hire_date` year NOT NULL,
  `title` enum('Adjunct Professor','Associate Professor','Assistant Professor','Professor','Professor of Practice','Lecturer','Senior Lecturer') NOT NULL,
  PRIMARY KEY (`uni`),
  CONSTRAINT `faculty_person` FOREIGN KEY (`uni`) REFERENCES `person_three` (`uni`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

CREATE TABLE `student_three` (
  `uni` varchar(12) NOT NULL,
  `major` varchar(4) NOT NULL,
  PRIMARY KEY (`uni`),
  CONSTRAINT `student_person` FOREIGN KEY (`uni`) REFERENCES `person_three` (`uni`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

Function

Switch to notebook.

```
CREATE DEFINER=`root`@`localhost` FUNCTION `generate_uni`(first_name VARCHAR(128),
    last_name VARCHAR(128)) RETURNS varchar(12) CHARSET utf8mb4
  DETERMINISTIC
BEGIN

  DECLARE result      VARCHAR(12);
  DECLARE fn_prefix   VARCHAR(2);
  DECLARE ln_prefix   VARCHAR(2);
  DECLARE uni_prefix  VARCHAR(5);
  DECLARE uni_count   INT;

  SET fn_prefix = SUBSTRING(first_name, 1, 2);
  SET ln_prefix = SUBSTRING(last_name, 1, 2);
  SET fn_prefix = LOWER(fn_prefix);
  SET ln_prefix = LOWER(ln_prefix);

  SET uni_prefix = CONCAT(fn_prefix, ln_prefix, '%');

  SET uni_count = (SELECT count(*) FROM person_three WHERE uni LIKE uni_prefix);

  SET result = CONCAT(fn_prefix, ln_prefix, uni_count);

RETURN result;
END
```

Design Pattern for Inheritance Integrity

- So far, we have created:
  - A three-table solution to a simple inheritance design problem.
  - Core elements are:
    - Function for computing UNIs.
    - A view to query the concrete class without exposing inheritance.
    - A procedure to create one of the concrete classes and the base classes.
- But,
  - We would still need to implement the update and delete stored procedures to ensure encapsulation and integrity. This prevents users from corrupting the data by removing the need to understand the model and write SQL scripts.
  - Implement a Trigger that rejects changing the UNI. Let's do that now.
- But, what if someone mistakenly uses the tables directly with SQL statements. That would still cause integrity problems.
- We can use identity, roles and permissions.

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** U_1, U_2, U_3
- **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_1, U_2, U_3

- <privilege-list> may be **all** to revoke all privileges the revokee may hold.
- If <revokee-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 grantees, 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.