W4111 – Introduction to Databases
Section 002, Spring 2021

Lecture 5: Advanced SQL and Projects

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

W4111 – Introduction to Databases
Section 002, Spring 2021

Lecture 5: Advanced SQL and Projects

We will start in a couple of minutes.

Contents

Contents

- Final SQL Concepts:
  - Constraints.
  - Indexes.
  - Metadata and Catalog (and an insight into HW2, programming).
  - ~~More Set Like Concepts~~ (Slides at end of presentation. Please just read).
  - Functions, Triggers, Procedures.

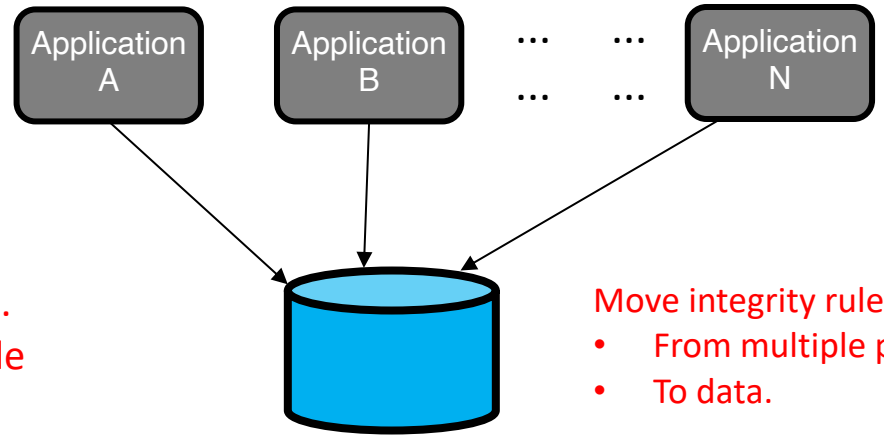
- Some Advanced ER → Relational Mapping
  - Multi-valued attributes.
  - Inheritance.
  - N-ary relationships, for N > 2.

- Starting to Understand HW3 and HW4 – The Projects:
  - Overview and concepts.
  - My project as an example.

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



Note:
- Integrating constraints as part of data definition is one of the major benefits.
- Eliminates the need to coordinate code in multiple applications.

Move integrity rules
- From multiple programs.
- To data.

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

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')))
```

Notes:

- I would make *semester* an ENUM.
- I normally keep check constraints relatively simple predicates on attributes.
- Not all engines support the more sophisticated features, e.g. subqueries.

Check Constraint and other Things

```
CREATE TABLE `university_faculty` (
  `guid` varchar(256) NOT NULL,
  `first_name` varchar(128) NOT NULL,
  `last_name` varchar(128) NOT NULL,
  `preferred_email` varchar(256) DEFAULT NULL,
  `title` enum('Adjunct Professor','Associate Professor','Assistant Professor','Professor','Professor of Practice','Lecturer','Senior Lecturer') NOT NULL,
  `hire_year` varchar(32) NOT NULL,
  `uni` varchar(12) NOT NULL,
  `IQ` int DEFAULT NULL,
  `uni_email` varchar(256) GENERATED ALWAYS AS (concat(`uni`,_utf8mb4'@columbia.edu')) STORED,
  `preferred_name` varchar(128) DEFAULT NULL,
  PRIMARY KEY (`guid`),
  UNIQUE KEY `uni_UNIQUE` (`uni`),
  UNIQUE KEY `uni_email_UNIQUE` (`uni_email`),
  CONSTRAINT `university_faculty_chk_1` CHECK ((((`IQ` > 0) and (`IQ` <= 200)) AND
    ((if(last_name='Ferguson',50,200) >= iq)))),
  CONSTRAINT `university_faculty_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;
```

Switch to notebook for example.

- Two simple check constraints.
- Two unique keys: uni cannot be NULL, preferred_email must be unique if not NULL.
- Auto-generated column: uni_email
- Example of an ENUM.

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**

Note:

- I do not use cascading, but that is just my preference.
- I believe that changes should be explicit, not implemented behind the scenes.

Integrity Constraint Violation During Transactions

- Consider:

  create table *person* (
  ID **char**(10),
  name **char**(40),
  mother **char**(10),
  father **char**(10),
  primary key *ID,*
  foreign key *father* **references** *person,*
  foreign key *mother* **references** *person*)

- How to insert a tuple without causing constraint violation?
  - Insert father and mother of a person before inserting person
  - OR, set father and mother to null initially, update after inserting all persons (not possible if father and mother attributes declared to be **not null**)
  - OR defer constraint checking

Note: Do not worry about transaction for now.

Complex Check Conditions

- The predicate in the check clause can be an arbitrary predicate that can include a subquery.

  check (*time_slot_id* **in** (**select** *time_slot_id* **from** *time_slot*))

  The check condition states that the time_slot_id in each tuple in the *section* relation is actually the identifier of a time slot in the *time_slot* relation.

  - The condition has to be checked not only when a tuple is inserted or modified in *section* , but also when the relation *time_slot* changes

Note:

- Not all databases support complex check conditions.
- You can use triggers, which we will cover, to implement more complex check constraints.

Assertions

- An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
- The following constraints, can be expressed using assertions:
- For each tuple in the *student* relation, the value of the attribute *tot_cred* must equal the sum of credits of courses that the student has completed successfully.
- An instructor cannot teach in two different classrooms in a semester in the same time slot
- An assertion in SQL takes the form:

  create assertion <assertion-name> **check** (<predicate>);

Note: Assertions are inconsistently supported and not that common.

User-Defined Types

- **create type** construct in SQL creates user-defined type

  create type *Dollars* **as numeric (12,2) final**

- Example:

  create table *department*
  (*dept_name* **varchar** (20),
  building **varchar** (15),
  budget Dollars);

Note:
Domains and UDTs are an important concept but not as well supported as they should be.

Domains

- **create domain** construct in SQL-92 creates user-defined domain types

  create domain *person_name* **char**(20) **not null**

- Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.
- Example:

  create domain *degree_level* **varchar**(10)
  constraint *degree_level_test*
  check (**value in** ('Bachelors', 'Masters', 'Doctorate'));

Note:
Support for domains is also inconsistent.

Defining Indexes

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);

Note:

- MySQL conflates and makes obscure concepts like Key, Constraint and Index.
- MySQL Indexes:
  - Primary
  - Unique

  } These are actually constraints. Indexes are necessary efficiently enforcing constraints.
  - Index
  - Full Index

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*

Complex Example
Columbia Courses, Sections, Enrollments
(Part I – Course)

Columbia Courses

(Example: ECON W1105 001 Principles of Economics, 4 pts)

	Example	Description
Call # (5 digit number)	***16238***	This 5 digit code is assigned to individual courses and is specific to each semester.
Department Code (4 letter code)	***ECON***	This 4 letter code represents the Academic Department that manages the course.
Course Number (1 capital letter followed by 4 digit code)	***W***	The *capital letters* indicate the instructor teaching the course and their affiliation with a **division, school** or **affiliate** of the **University**. Unless otherwise noted, courses numbers beginning with the following letters are generally open to CC/SEAS undergraduate students: BC Barnard College C Columbia College E Engineering and Applied Science F General Studies G Graduate School of Arts and Sciences V Interschool course with Barnard W Interfaculty course X Barnard College
	1105	The *first digit* indicates the **level of the course**. Generally, levels are indicated as: 0 Course that cannot be credited toward any degree 1 Undergraduate course, introductory 2 Undergraduate course, intermediate 3 Undergraduate course, advanced 4 Graduate course that is open to qualified undergraduates 6 Graduate course 8 Graduate course, advanced 9 Graduate research course or seminar
Course Section	***001***	Based on course demand, some academic departments offer the same course during 2 or more different time slots. Each time slot is assigned a different section number.
Points/Credits	***4***	The term "points" and "credits" are often used interchangeably and is generally related to the number of classroom contact hours.

University Course Number Codes

A	*Architecture, Planning, and Preservation**
B	*Business**
BC	**Barnard College**
C	**Columbia College**
D	*Dentistry***
E	**Engineering and Applied Science**
F	**General Studies**
G	**Graduate School of Arts and Sciences**
H	*Reid Hall, Paris***
I	*Berlin Consortium Program***
J	*Journalism**
K	*Continuing Education***
L	*Law***
M	*Medicine***
N	*Nursing***
O	*Union Theological***
P	*School of Public Health**
R	*School of the Arts**
S	**Summer Session**
T	*Social Work**
TA–TZ	*Teachers College**
U	*International and Public Affairs**
V	**Interschool course with Barnard**
W	**Interfaculty course**
X	**Barnard College**
Z	*American Language Program(no credit)***

Columbia Courses

(Example: ECON W1105 001 Principles of Economics, 4 pts)

	Example	Description
Call # (5 digit number)	***16238***	This 5 digit code is assigned to individual courses and is specific to each semester.
Department Code (4 letter code)	***ECON***	This 4 letter code represents the Academic Department that manages the course.
Course Number (1 capital letter followed by 4 digit code)	***W***	The *capital letters* indicate the instructor teaching the course and their affiliation with a **division, school** or **affiliate** of the **University**. Unless otherwise noted, courses numbers beginning with the following letters are generally open to CC/SEAS undergraduate studen
	1105	The *first digit* indicates the **level of the course**. Generally, levels are indicated as:
Course Section	***001***	Based on course demand, some academic departments offer the same course during 2 or more different time slots. Each time slot is assigned a different section number.
Points/Credits	***4***	The term "points" and "credits" are often used interchangeably and is generally related to the number of classroom contact hours.

BC	Barnard College
C	Columbia College
E	Engineering and Applied Science
F	General Studies
G	Graduate School of Arts and Sciences
V	Interschool course with Barnard
W	Interfaculty course
X	Barnard College

0	Course that cannot be credited toward any degree
1	Undergraduate course, introductory
2	Undergraduate course, intermediate
3	Undergraduate course, advanced
4	Graduate course that is open to qualified undergraduates
6	Graduate course
8	Graduate course, advanced
9	Graduate research course or seminar

University Course Number Codes

A	*Architecture, Planning, and Preservation**
B	*Business**
M	*Medicine***
N	*Nursing***
O	*Union Theological***
P	*School of Public Health**
R	*School of the Arts**
S	**Summer Session**
T	*Social Work**
TA–TZ	*Teachers College**
U	*International and Public Affairs**
V	**Interschool course with Barnard**
W	**Interfaculty course**
X	**Barnard College**
Z	*American Language Program(no credit)***

I am going to ignore the more complex table of instructor affiliation codes, and use the simple set.

Course Number

```
CREATE TABLE `courses` (
  `dept_code` char(4) NOT NULL,
  `faculty_code` enum('BC','C','E','F','G','V','W','X') NOT NULL,
  `level` enum('0','1','2','3','4','6','8','9') NOT NULL,
  `number` char(3) NOT NULL,
  `title` varchar(256) NOT NULL,
  `description` varchar(1204) NOT NULL,
  `course_number` varchar(64) GENERATED ALWAYS AS (concat(`dept_code`,`faculty_code`,`level`,`number`)) STORED NOT NULL,
  `credits` int NOT NULL,
  PRIMARY KEY (`course_number`),
  KEY `dept_idx` (`dept_code`),
  KEY `full_number_idx` (`faculty_code`,`level`,`number`),
  KEY `dept_and_level` (`dept_code`,`level`),
  KEY `dept_and_credit` (`dept_code`,`credits`),
  CONSTRAINT `courses_chk_1` CHECK ((`number` between _utf8mb4'000' and _utf8mb4'999')),
  CONSTRAINT `courses_chk_2` CHECK (((`credits` >= 0) and (`credits` <= 4)))
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

Switch to notebook.

Some observations:

- faculty_code and level are ENUMs, which you declare as strings.
- There is a check constraint that ensures that the last 3 digits of the number are between ‘000’ and ‘999’, and a constraint on credits.
- We always generate the full course_number. We store the value (versus compute on demand) → it can be a key.
- Will talk about indexes in a minute.

MySQL Indexes

- The MySQL Index Types are: Primary, Index, Unique, Full Text.
- Comment 1: This conflates indexes for efficiency and constraints for integrity, e.g. Primary, Unique.
- What is this Full Text index of which you speak?
  - ALTER TABLE `aaaa_simple_university`.`courses`
    ADD FULLTEXT INDEX `key_words` (`title`, `description`) VISIBLE;
- Enables querying for courses that contain a specific word or words.
- Note:

Go to notebook

  - This is NOT how large-scale text search works.
  - The search is on whole words, not partial words → Does not support autocomplete.
  - You can define “stop words” that the query/index ignores, e.g. “the”, “it”, ...
    https://dev.mysql.com/doc/refman/8.0/en/fulltext-stopwords.html

Metadata and Catalog

Metadata and Catalog

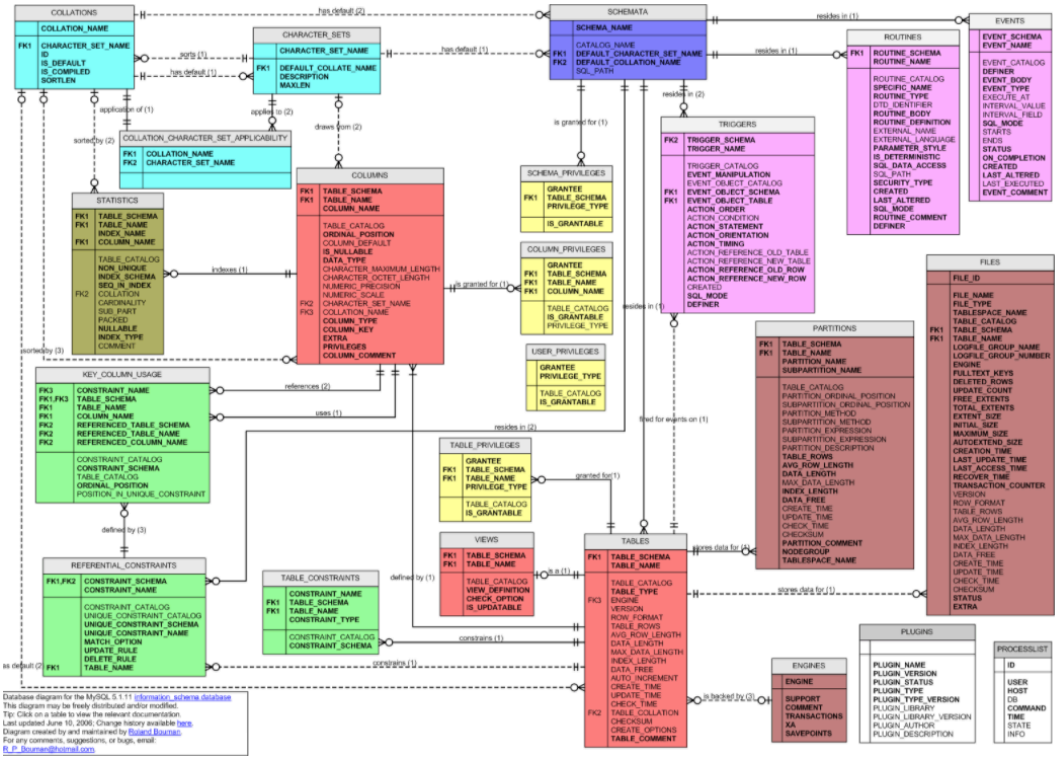
- ‘Metadata is "data that provides information about other data". In other words, it is "data about data". Many distinct types of metadata exist, including descriptive metadata, structural metadata, administrative metadata, reference metadata and statistical metadata.’ (https://en.wikipedia.org/wiki/Metadata)

- “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.

MySQL Catalog (Information_Schema)



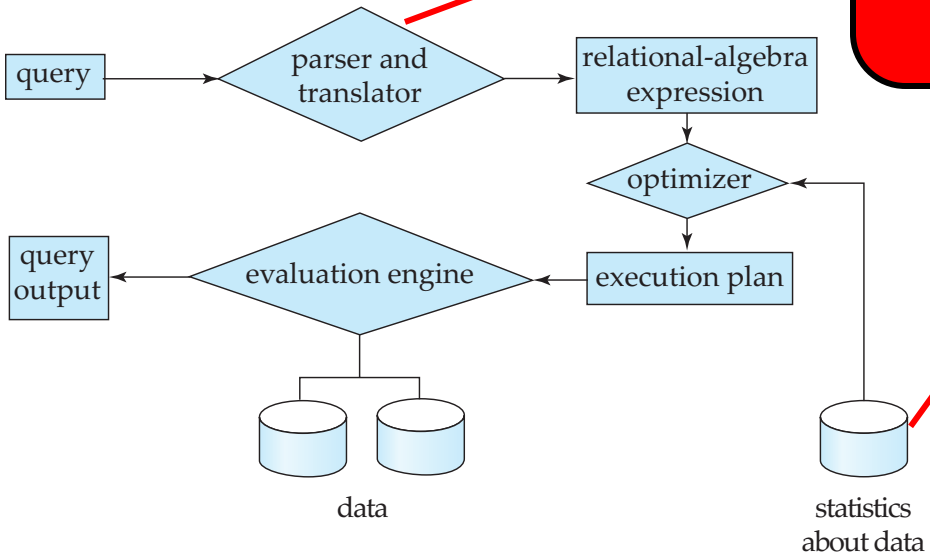
Some of the MySQL Information Schema Tables:

- '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.

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation

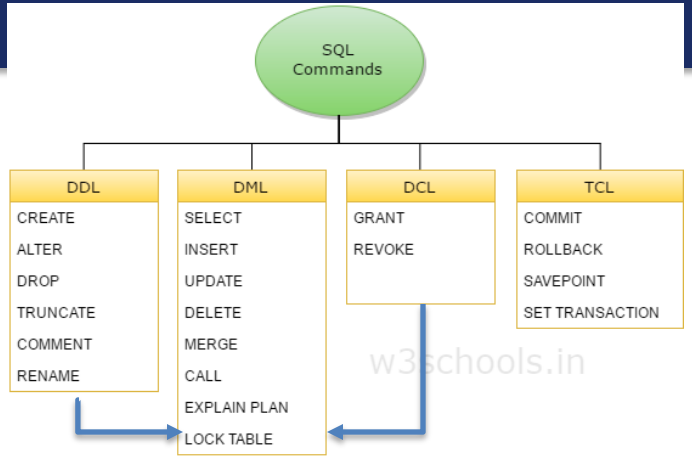


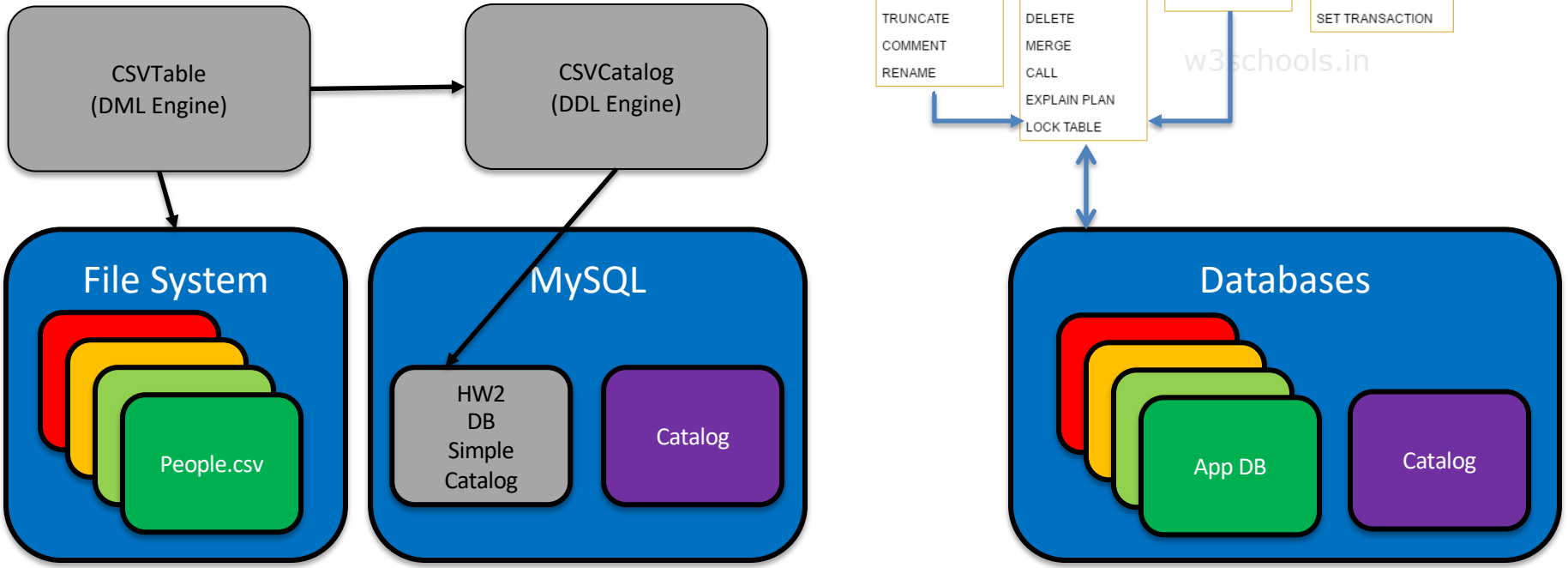
Database Catalog
- Tables, columns, types, ... needed to check syntax.
- Statistics used for cost based optimization and plans.

Switch to notebook.

DDL and DML (and HW2)

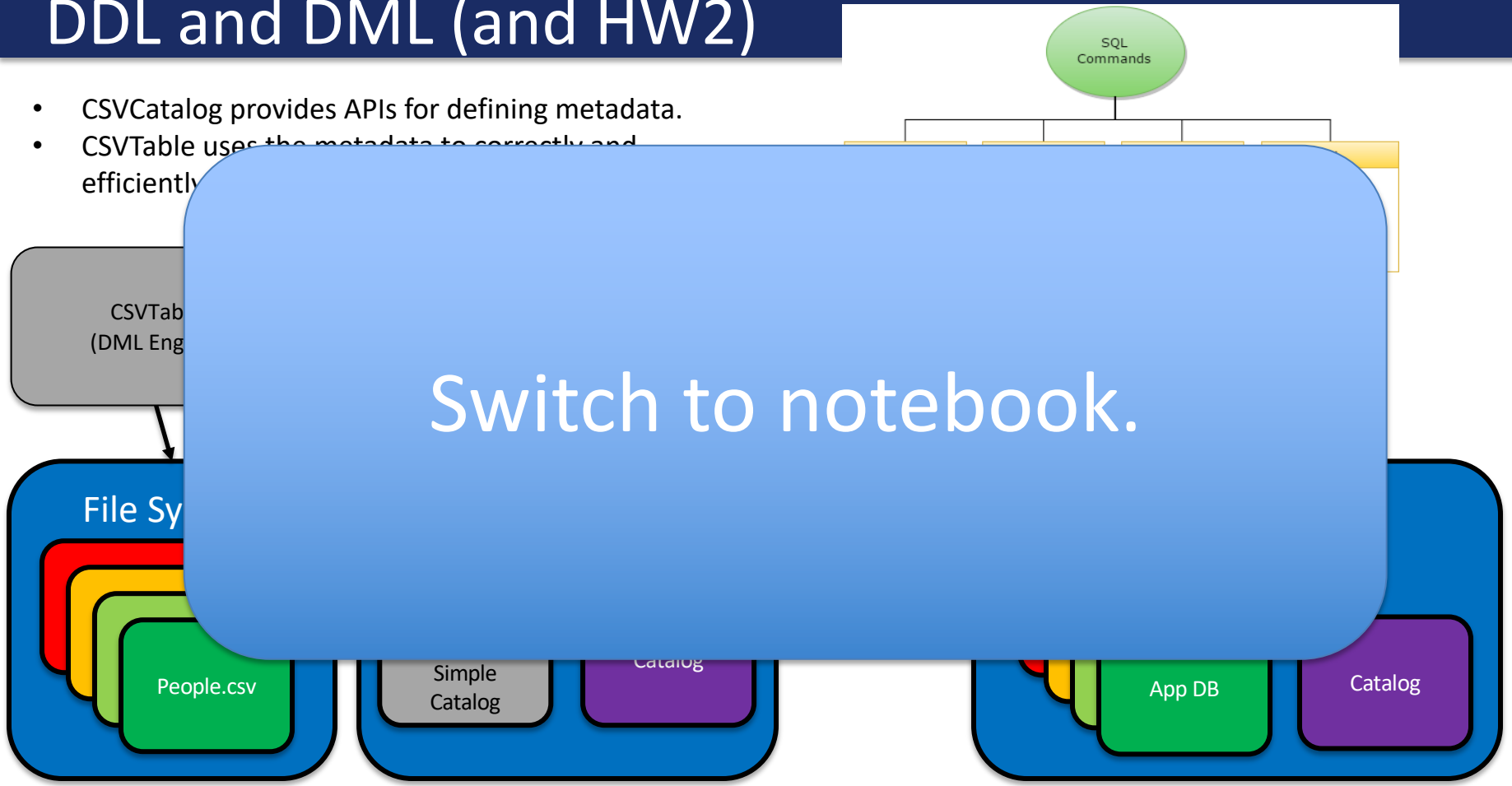
- CSVCatalog provides APIs for defining metadata.
- CSVTable uses the metadata to correctly and efficiently support find_by_template, join, insert, ...





DDL and DML (and HW2)

- CSVCatalog provides APIs for defining metadata.
- CSVTable uses the metadata to correctly and efficiently



Switch to notebook.

Functions, Procedures, Triggers

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 that 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'))
```

Switch to Notebook

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 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.

Procedures

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* <> '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

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

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.

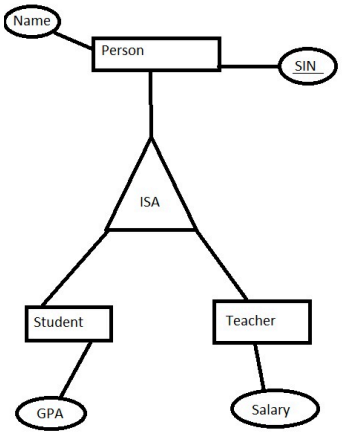
Complex Example
Columbia Courses, Sections, Enrollments
(Part I – Students and Faculty)

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



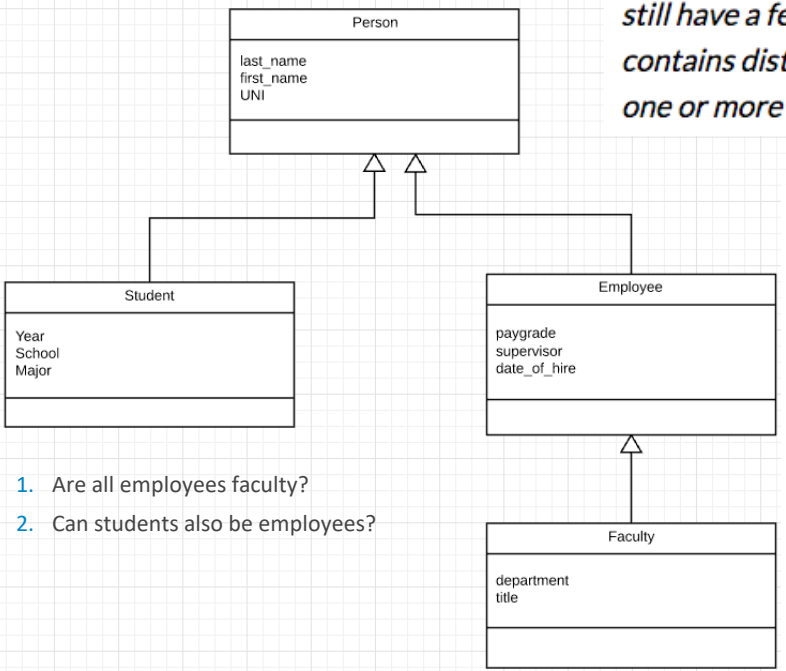
- "In object-oriented programming, inheritance is the mechanism of basing an object or class upon another object (prototype-based inheritance) or class (class-based inheritance), retaining similar implementation. Also defined as deriving new classes (sub classes) from existing ones such as super class 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.

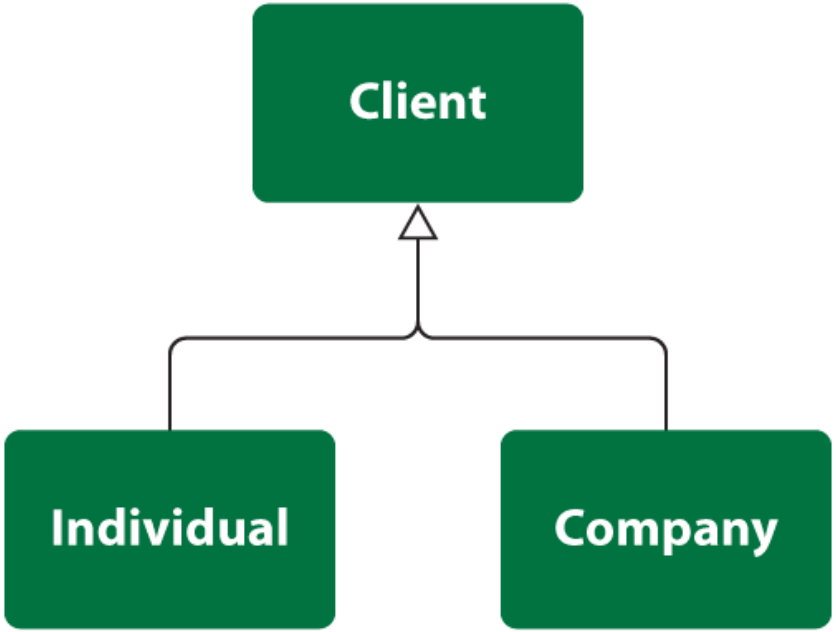
2 disjoint/overlapping

- 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).

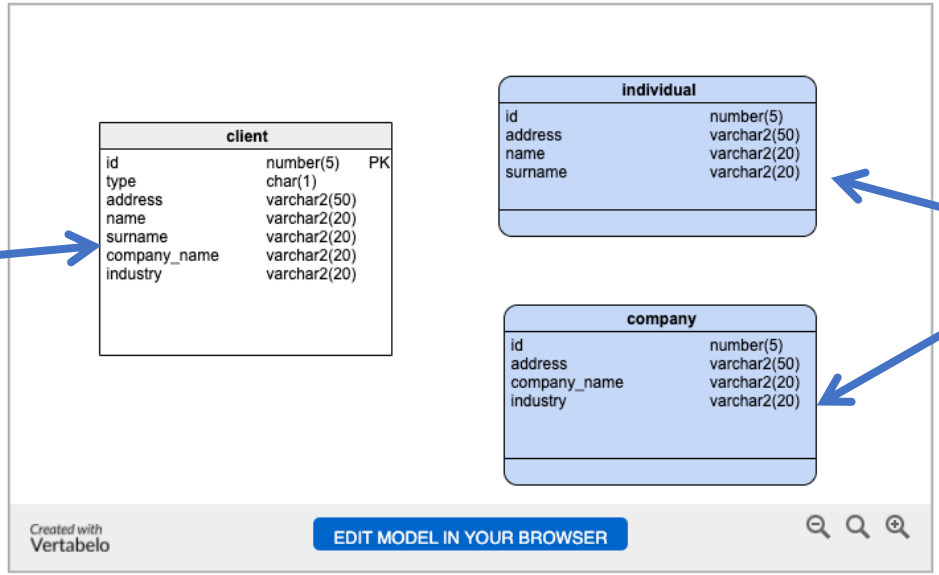


One Table Implementation

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

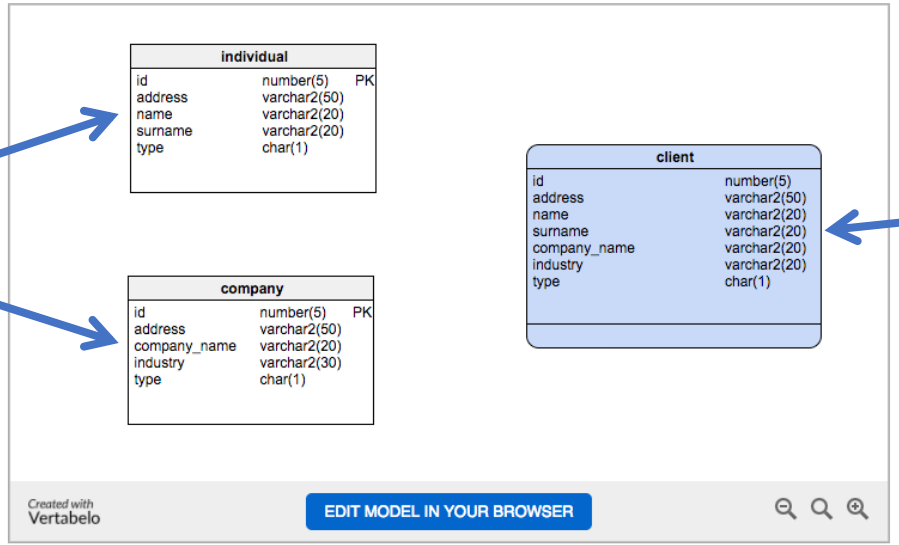
Views

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

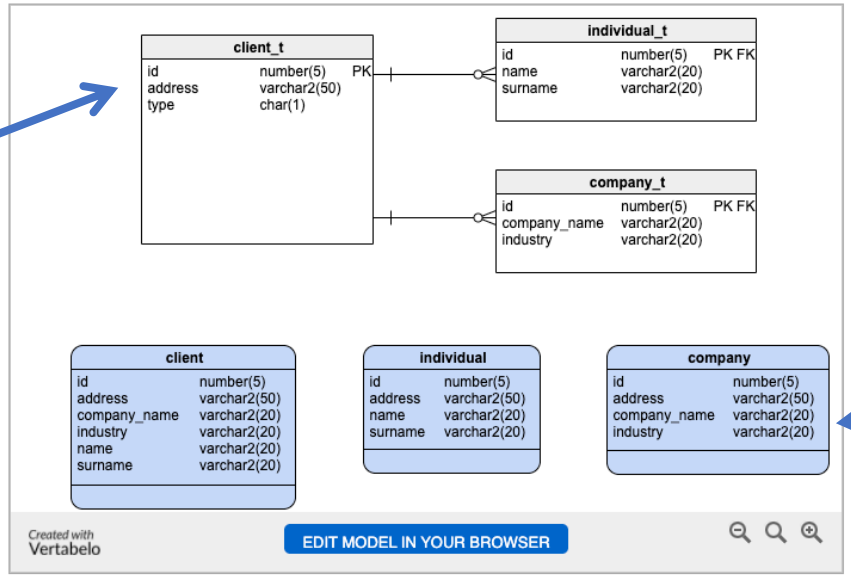
View

Two Table Implementation

Three-table implementation

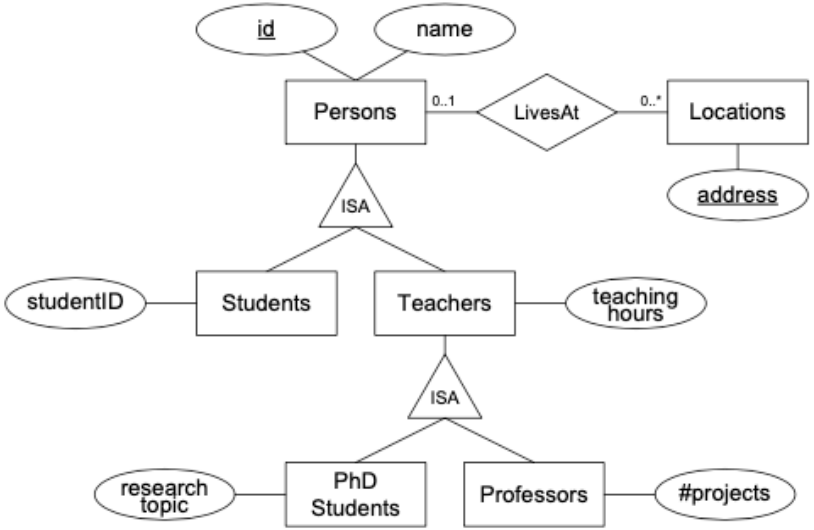
In a third solution we create a single table `client_t` for the parent table, containing common attributes for all subtypes, and tables for each subtype (`individual_t` and `company_t`) where the primary key in `client_t` (base table) determines foreign keys in dependent tables. There are three views: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

Tables

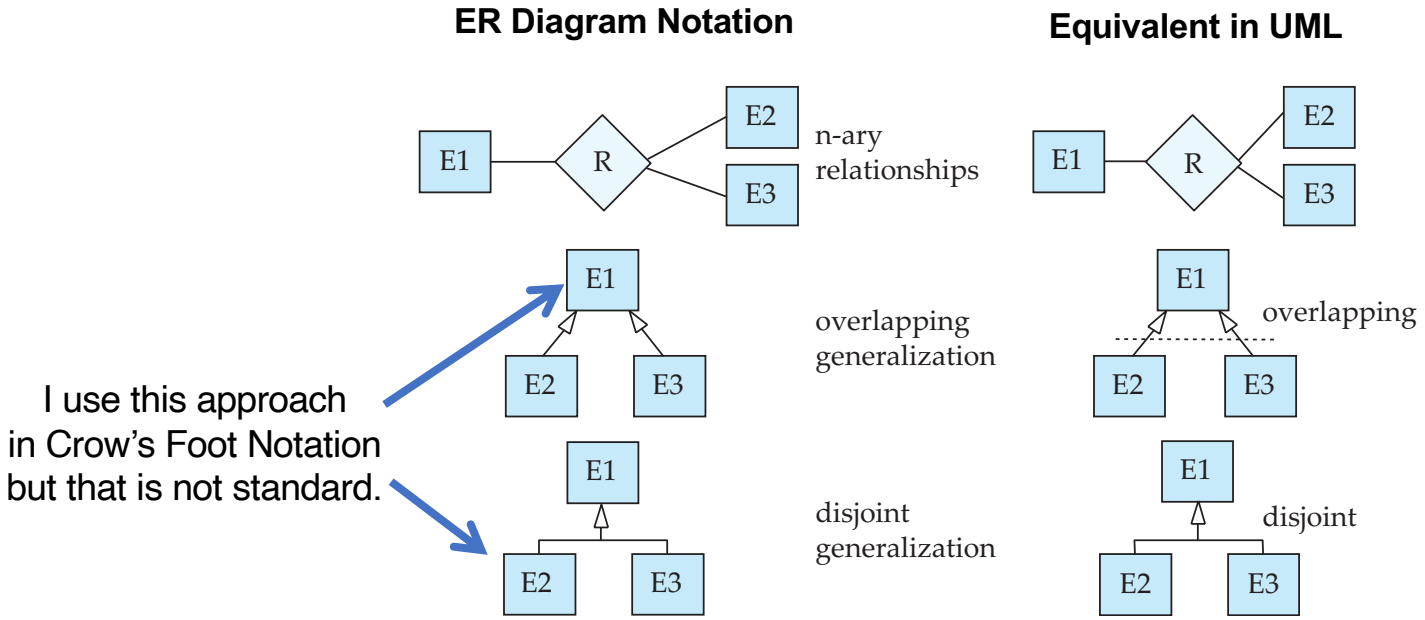


Views

ISA Relationship



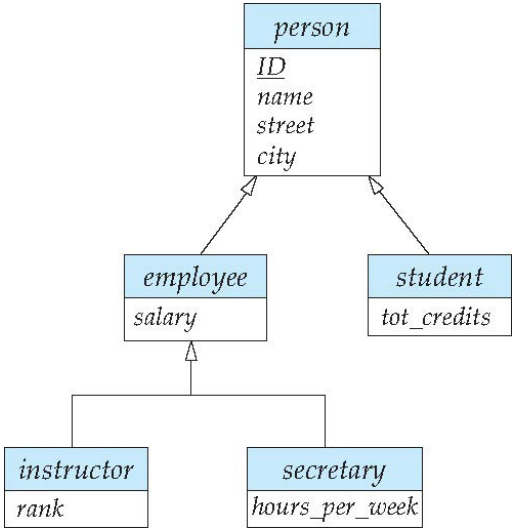
ER vs. UML Class Diagrams



* 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

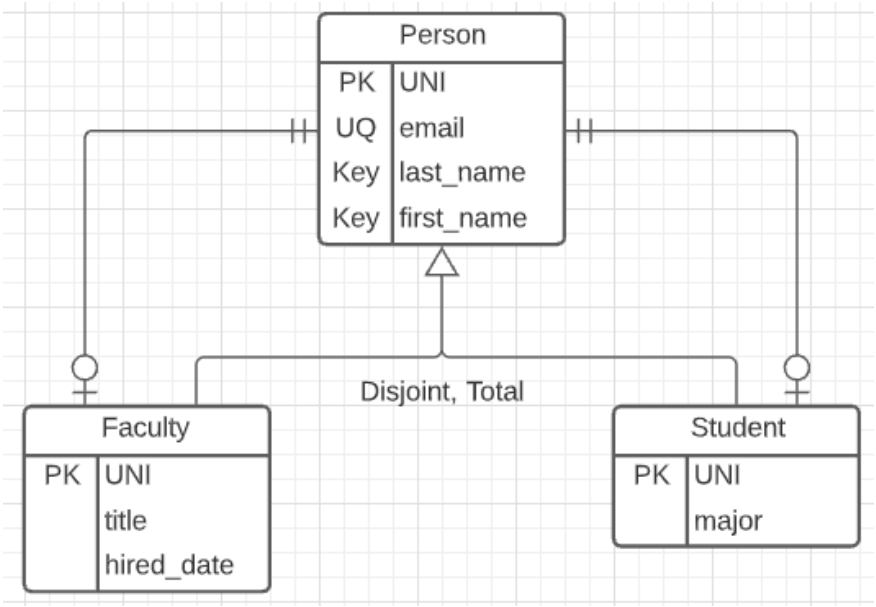
- **Completeness constraint** -- specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within a generalization.
  - **total**: an entity must belong to one of the lower-level entity sets
  - **partial**: an entity need not belong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

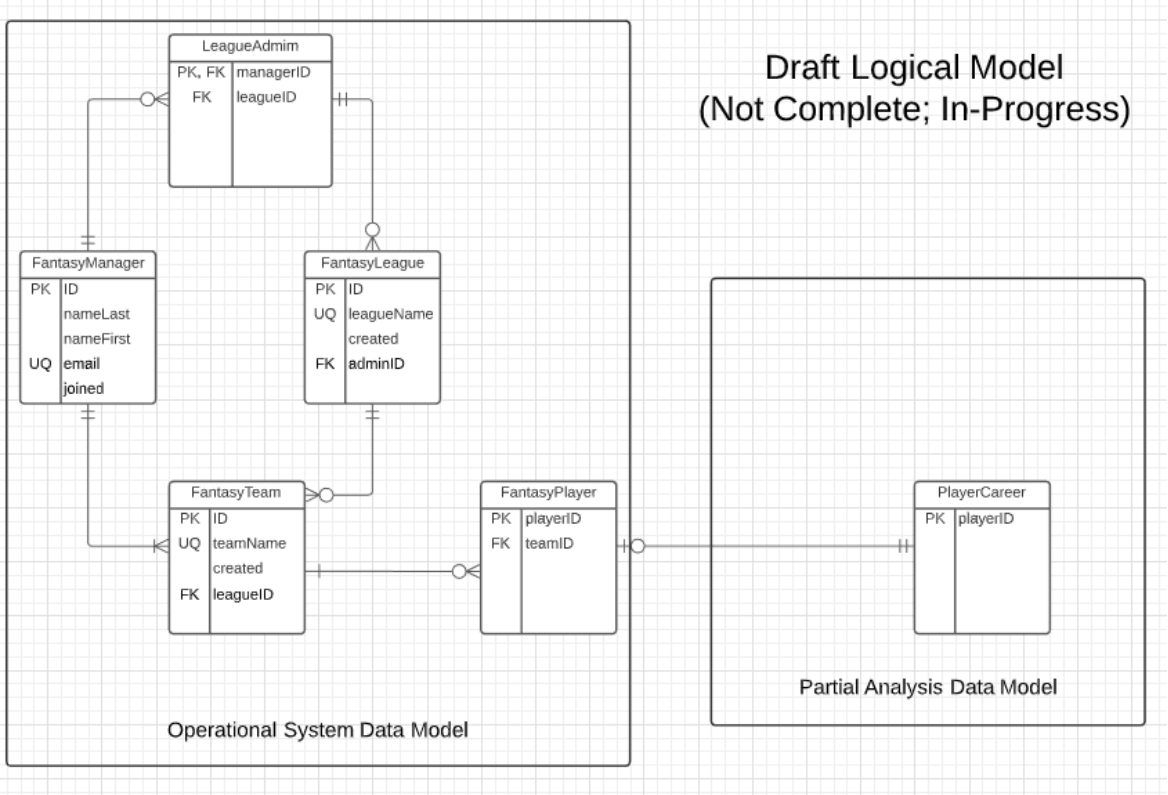
```
CREATE DEFINER=`root`@`localhost` FUNCTION `generate_uni`(first_name VARCHAR(128),
    last_name VARCHAR(128)) RETURNS varchar(12) CHARSET utf8mb4
  DETERMINISTIC
BEGIN

  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
```

Projects: HW3 and HW4

Pretend to Play Fantasy Baseball

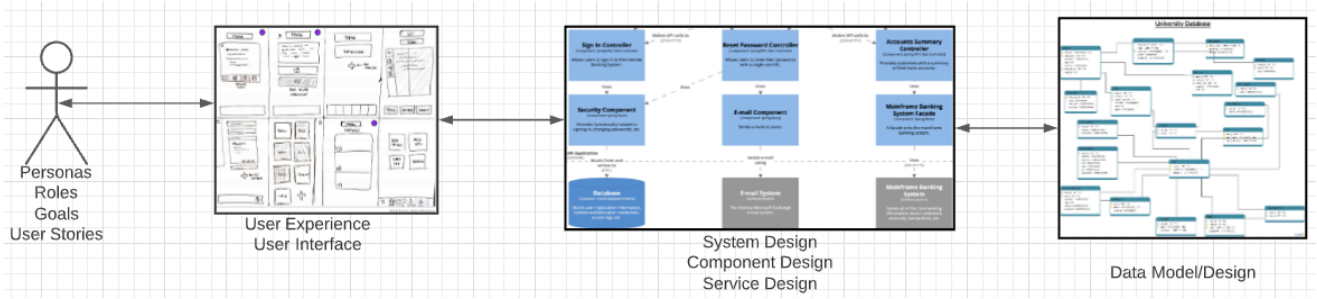
- My example projects for the two tracks will be a **simple** fantasy baseball solution.
  - "**Fantasy baseball** is a game in which people manage rosters of league baseball players, either online or in a physical location, using fictional fantasy baseball team names. The participants compete against 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.

In-Progress Operation System Data Model



Problem Statement – Modified Reminder

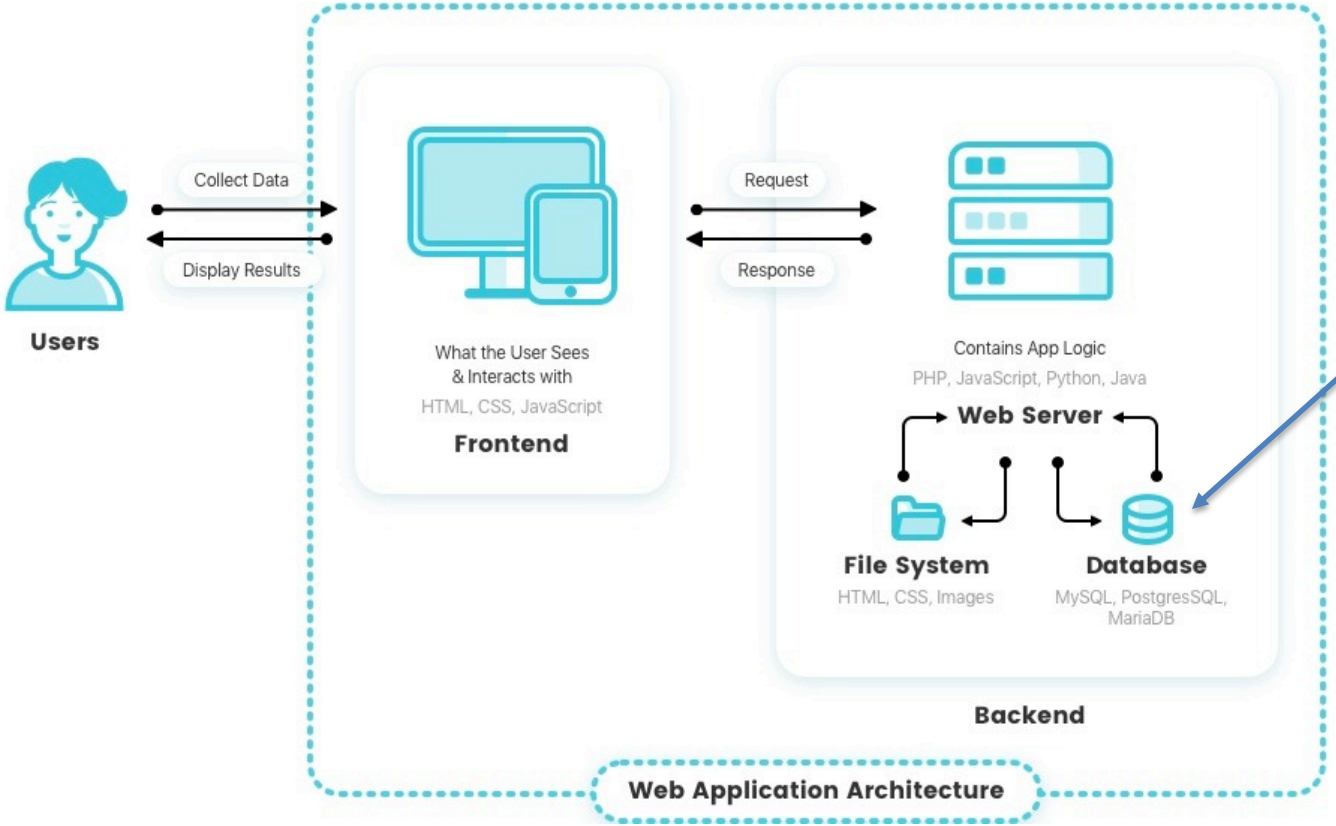
- We must build a system that supports academic operations in a university.
- We will design, develop, test and deploy the system iteratively and continuously.
- There are four core domains.



- 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. (Will only cover this now).

Web Application

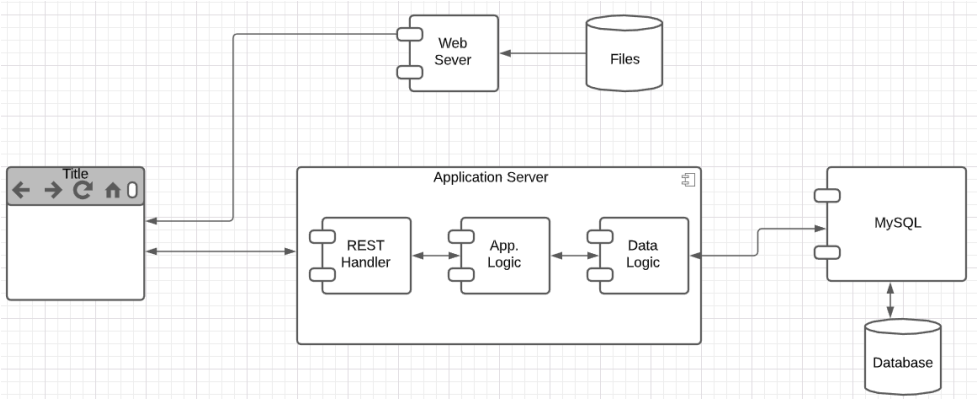


IMDB Database

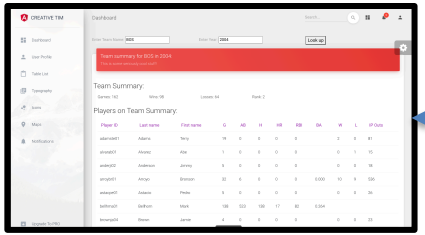
It would be "cool" if we could look at the data directly!

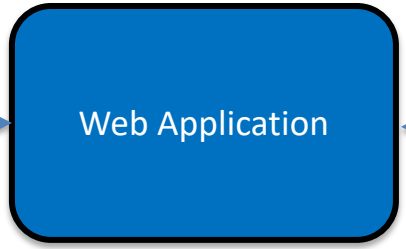
User Story

- "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.

Switch to Notebook

Switch to Notebook
and
Code.

- Angular project
- OpenAPI
- Application code

Material to Read

Some More Set-Like Concepts

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 <comp> **some** $r \Leftrightarrow \exists\ t \in r$ such that (F <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 ≠ 5)

(= **some**) ≡ **in**
However, (≠ **some**) ≢ **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 <comp> **all** $r \Leftrightarrow \forall\ t \in r$ (F <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 ≠ 4 and 5 ≠ 6)

(≠ **all**) ≡ **not in**

However, (= **all**) ≠ **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);
```

Completing Subqueries

- A subquery is a query inside a query.
- Logically, the engine calls the subquery when evaluating every row in processing.
- Correlation means properties from the outer row being evaluated are inputs to the inner query.
- Subqueries:
    - Return tables if in the FROM clause.
    - Scalars in the SELECT clause.
    - Tables or scalars depending on the comparison operator in WHERE.

Subqueries in the From 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;
```

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

Switch to Notebook

Backup