W4111 – Introduction to Databases
Section 002, Fall 2021

Lecture 6: ER, Relational, SQL (Final)

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

Contents

Contents

- Midterm overview.
- Questions, answers, discussion.
- Codd's 12 Rules revisited.
- SQL/RDB:
  - Metadata, DDL, catalog
  - Index definitions
  - Functions, procedures, triggers.
  - Security, grants, ...
  - Recursive queries
- Advanced ER Concepts
  - Inheritance
  - Aggregation
- A worked example involving: Advanced ER, functions/triggers, authorization, ...

Midterm Overview

Questions, Answers, Discussion

Codd's 12 Rules Revisited

Codd's 12 Rules

Rule 1: Information Rule

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

Rule 2: Guaranteed Access Rule

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

Rule 3: Systematic Treatment of NULL Values

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

Rule 4: Active Online Catalog

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

Rule 5: Comprehensive Data Sub-Language Rule

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

Rule 6: View Updating Rule

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

Codd's 12 Rules

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

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

Rule 8: Physical Data Independence

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

Rule 9: Logical Data Independence

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

Rule 10: Integrity Independence

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

Rule 11: Distribution Independence

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

Rule 12: Non-Subversion Rule

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

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.

Data Definition Language (DDL)

- Specification notation for defining the database schema

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

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

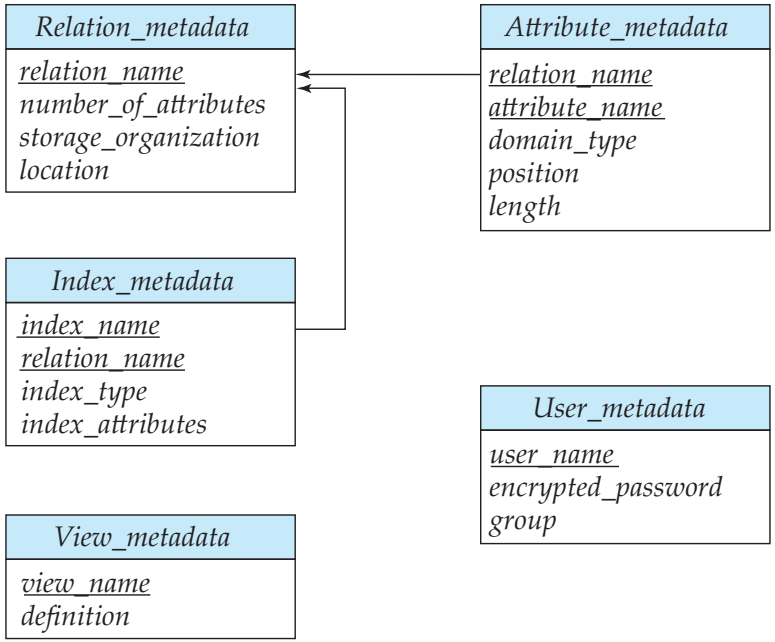
Data Dictionary Storage

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

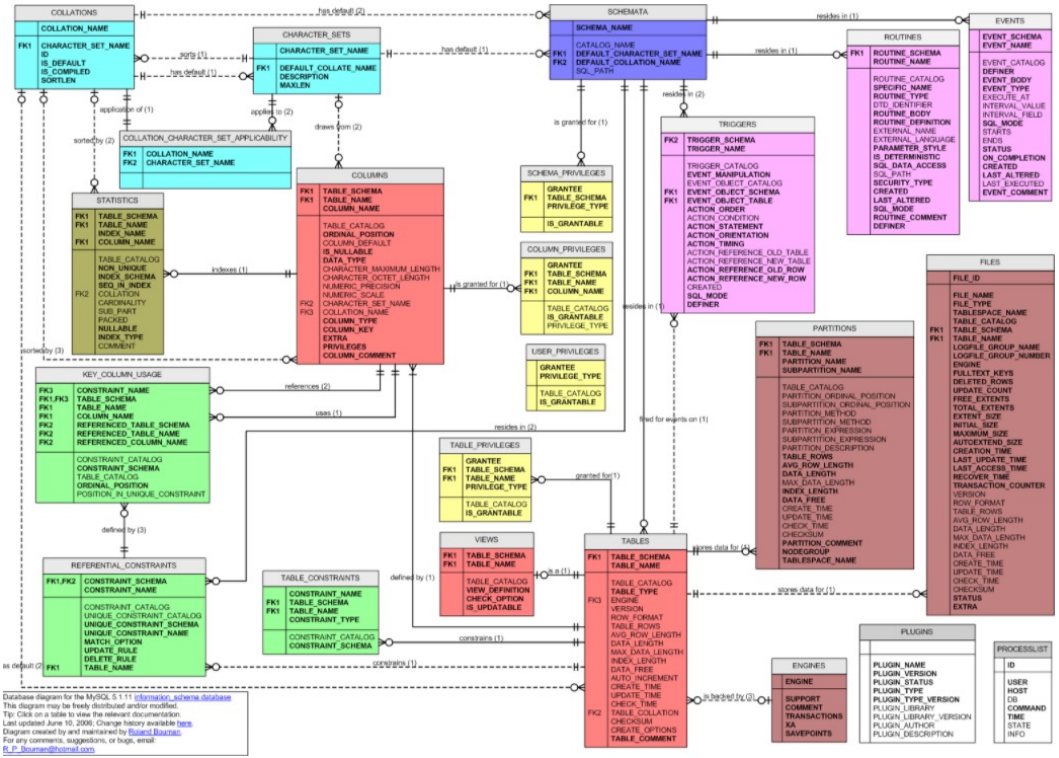
- Information about relations
  - names of relations
  - names, types and lengths of attributes of each relation
  - names and definitions of views
  - integrity constraints
- User and accounting information, including passwords
- Statistical and descriptive data
  - number of tuples in each relation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

Relational Representation of System Metadata

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



MySQL Catalog (Information_Schema)



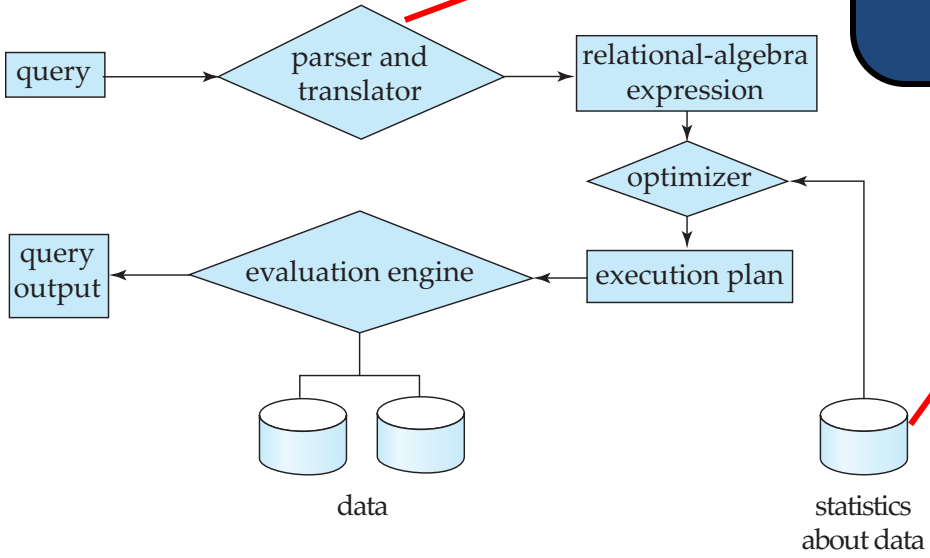
Some of the MySQL Information Schema Tables:

- 'ADMINISTRABLE_ROLE_AUTHORIZATIONS'
- 'APPLICABLE_ROLES'
- 'CHARACTER_SETS'
- 'CHECK_CONSTRAINTS'
- 'COLUMN_PRIVILEGES'
- 'COLUMN_STATISTICS'
- 'COLUMNS'
- 'ENABLED_ROLES'
- 'ENGINES'
- 'EVENTS'
- 'FILES'
- 'KEY_COLUMN_USAGE'
- 'PARAMETERS'
- 'REFERENTIAL_CONSTRAINTS'
- 'RESOURCE_GROUPS'
- 'ROLE_COLUMN_GRANTS'
- 'ROLE_ROUTINE_GRANTS'
- 'ROLE_TABLE_GRANTS'
- 'ROUTINES'
- 'SCHEMA_PRIVILEGES'
- 'STATISTICS'
- 'TABLE_CONSTRAINTS'
- 'TABLE_PRIVILEGES'
- 'TABLES'
- 'TABLESPACES'
- 'TRIGGERS'
- 'USER_PRIVILEGES'
- 'VIEW_ROUTINE_USAGE'
- 'VIEW_TABLE_USAGE'
- 'VIEWS'

- CREATE and ALTER statements modify the data.
- DBMS reads information:
  - Parsing
  - Optimizer
  - etc.

Usage Example: 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.

Demo

- Switch to notebook for some queries.

Index Definitions

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);

- What the statement means is “reference in the WHERE clause.”
- The “SELECT X,Y, ...” is (mostly) not relevant.

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*

Switch to Notebook for Examples

Some notes:

- An index may be on one columns or several columns together.
- For example,

```
ALTER TABLE `lahmansbaseballdb`.`people`
        ADD INDEX `both_names` (`nameLast` ASC, `nameFirst` ASC) VISIBLE;
```

  - This creates an index on (nameLast, nameFirst), which
    - Also accelerates queries that have just nameLast
    - But does not accelerate queries that have just nameFirst
    - We will see why in Module II
- Indexes support queries with =, % (pattern), >, >=, <, <=, BETWEEN, ...
- Indexes also work for string prefixes in some cases, e.g.
  - Works for *nameLast like 'Ferg%'* but not for *nameLast like '%Ferg'*
  - Works for *nameLast > 'Ferg'*
- Many databases, including MySQL automatically create indexes to accelerate processing primary key, unique and foreign key constraints.

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'))
```

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.

Triggers

Triggers

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.
- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed.
  - Specify the actions to be taken when the trigger executes.
- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.
  - Syntax illustrated here may not work exactly on your database system; check the system manuals

Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes
  - For example, **after update of** *takes* **on** *grade*
- Values of attributes before and after an update can be referenced
  - **referencing old row as** : for deletes and updates
  - **referencing new row as** : for inserts and updates
- Triggers can be activated before an event, which can serve as extra constraints. For example, convert blank grades to null.

```
create trigger setnull_trigger before update of takes
referencing new row as nrow
for each row
    when (nrow.grade = ' ')
begin atomic
        set nrow.grade = null;
end;
```

Trigger to Maintain credits_earned value

- **create trigger** *credits_earned* **after update of** *takes* **on** (*grade*)
  referencing new row as *nrow*
  referencing old row as *orow*
  for each row
  when *nrow.grade* <> '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.

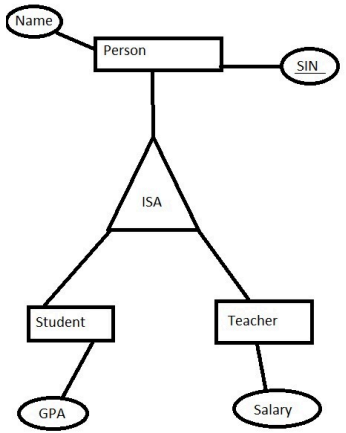
Complex Example
Columbia Courses, Sections, Enrollments
(Part I – Students and Faculty)

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

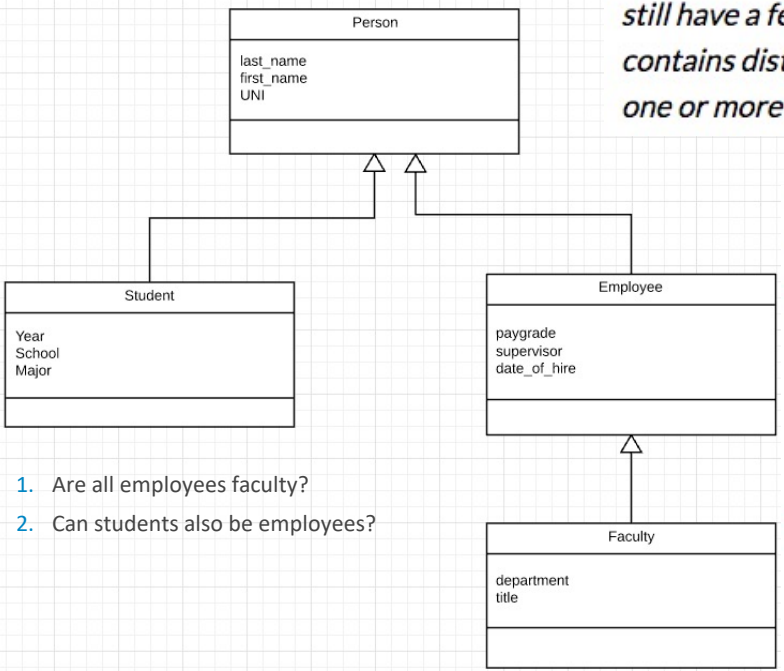


- "In object-oriented programming, inheritance is the mechanism of basing an object or class upon another object (prototype-based inheritance) or class (class-based inheritance), retaining similar implementation. Also defined as deriving new classes (sub classes) from existing ones such as super class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



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

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.

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.

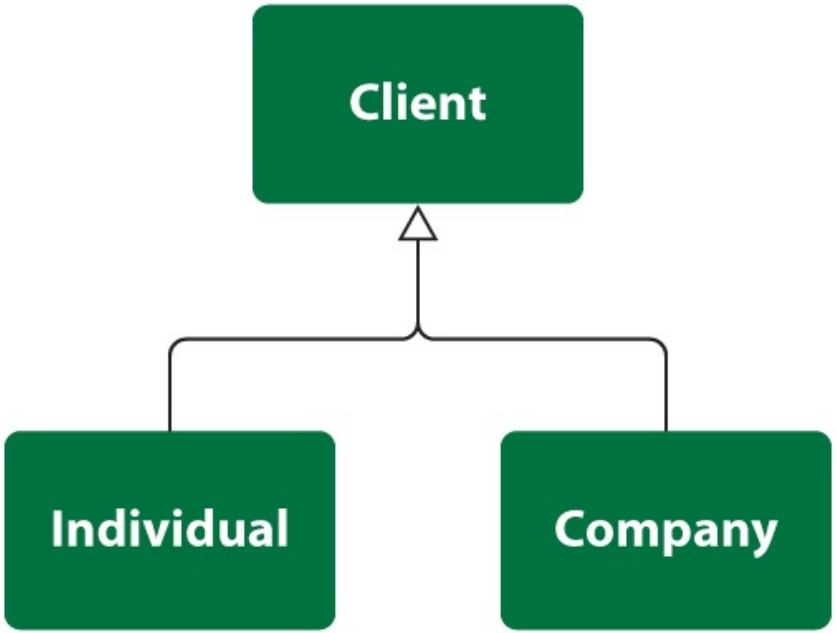
2 disjoint/overlapping

- 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).

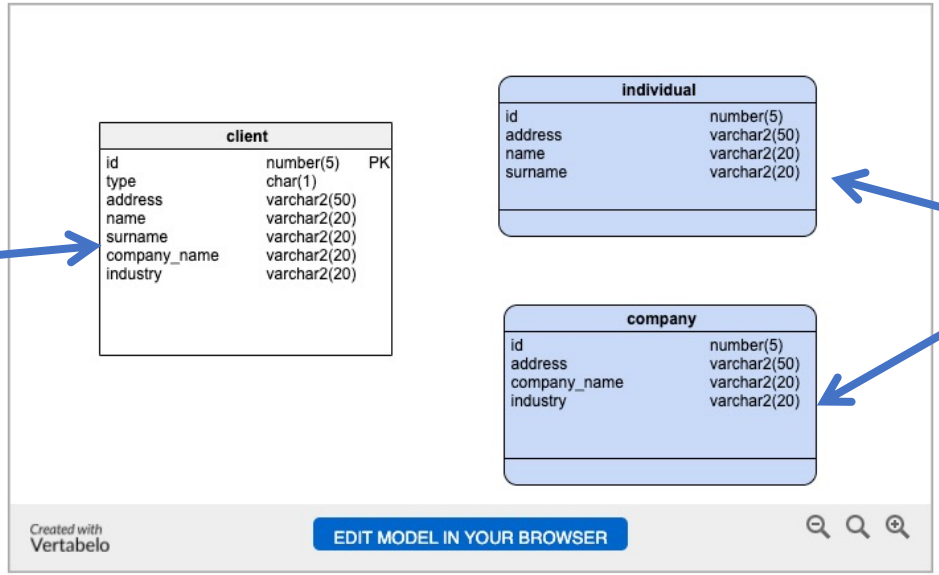


One Table Implementation

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

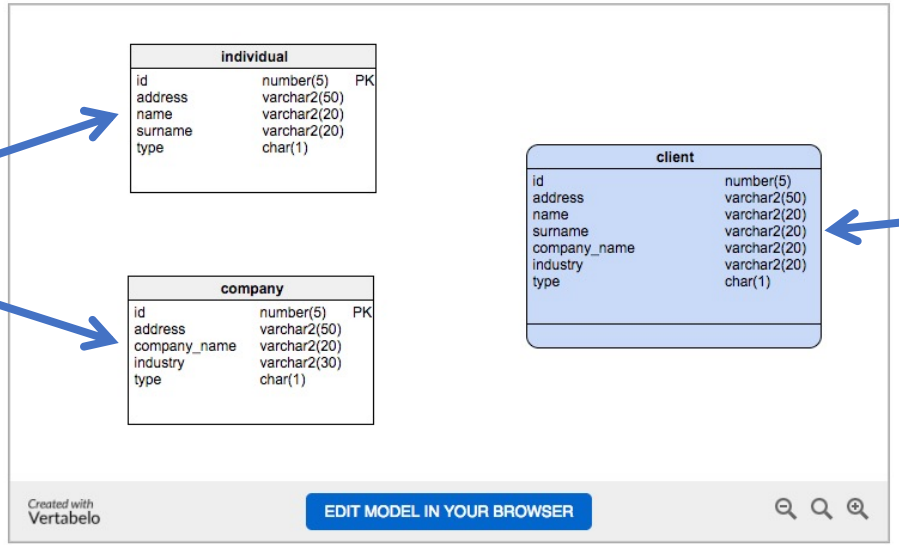
Views

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`.



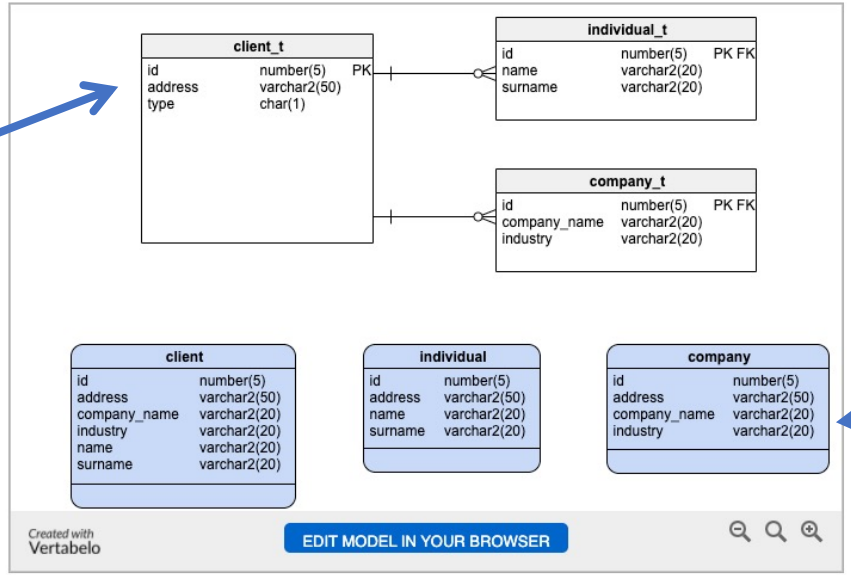
Tables

View

Two Table Implementation

Three-table implementation

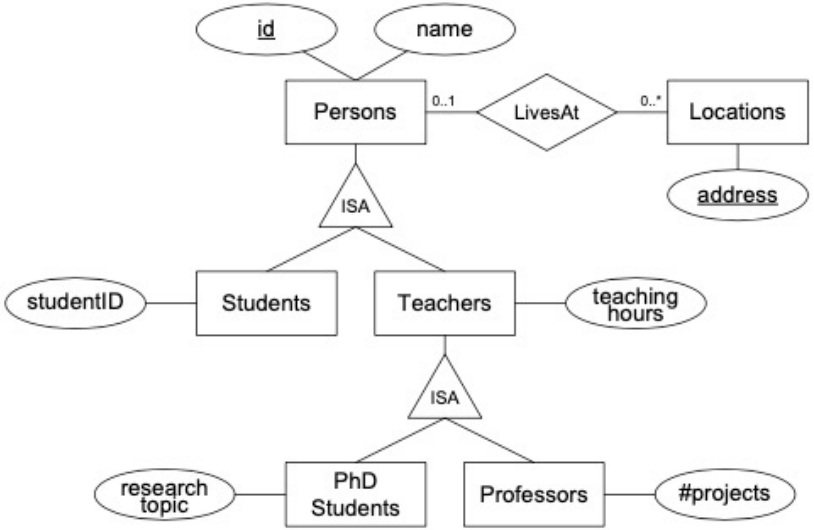
In a third solution we create a single table `client_t` for the parent table, containing common attributes for all subtypes, and tables for each subtype (`individual_t` and `company_t`) where the primary key in `client_t` (base table) determines foreign keys in dependent tables. There are three views: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



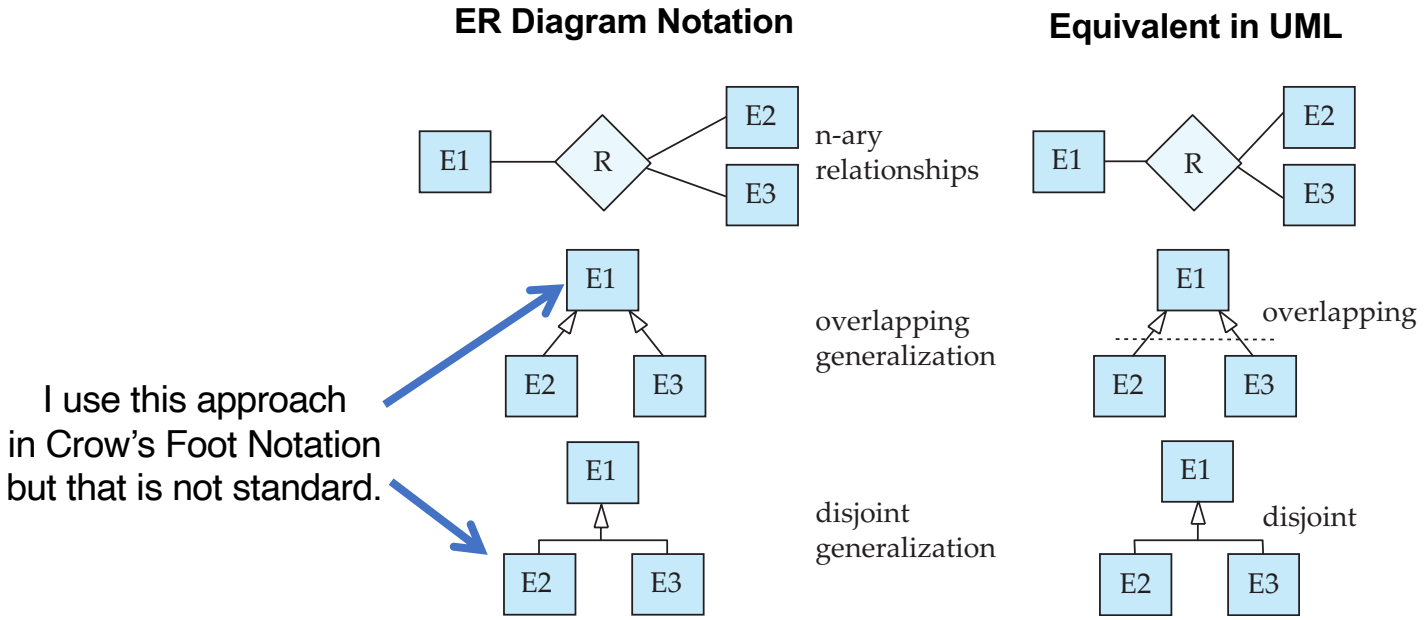
Tables

Views

ISA Relationship



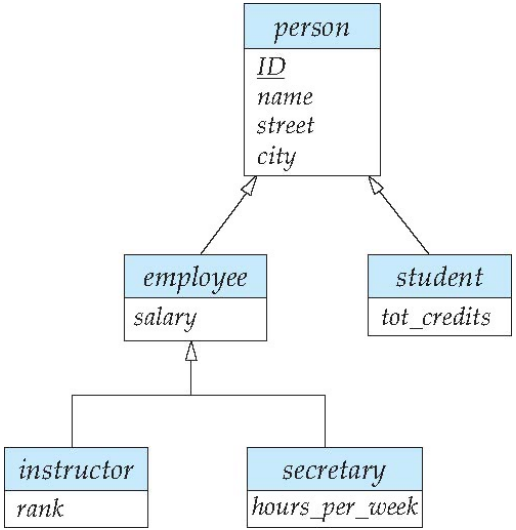
ER vs. UML Class Diagrams



* 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

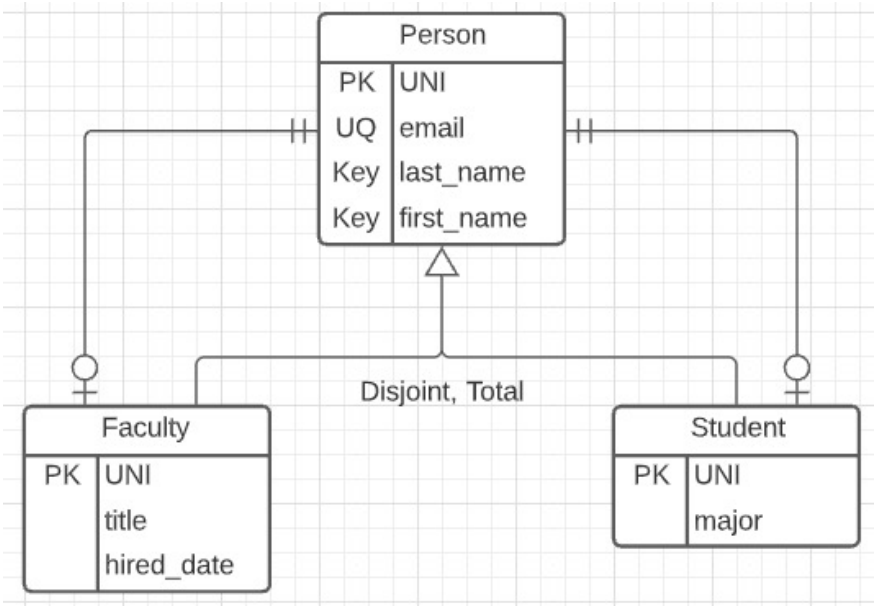
- **Completeness constraint** -- specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within a generalization.
  - **total**: an entity must belong to one of the lower-level entity sets
  - **partial**: an entity need not belong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

Some Advanced ER Concepts

Weak Entity Sets

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

Weak Entity Sets (Cont.)

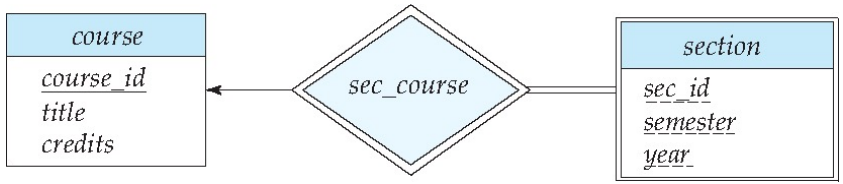
- An alternative way to deal with this redundancy is to not store the attribute *course_id* in the *section* entity and to only store the remaining attributes *section_id*, *year*, and *semester.*
  - However, the entity set *section* then does not have enough attributes to identify a particular *section*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

Weak Entity Sets (Cont.)

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

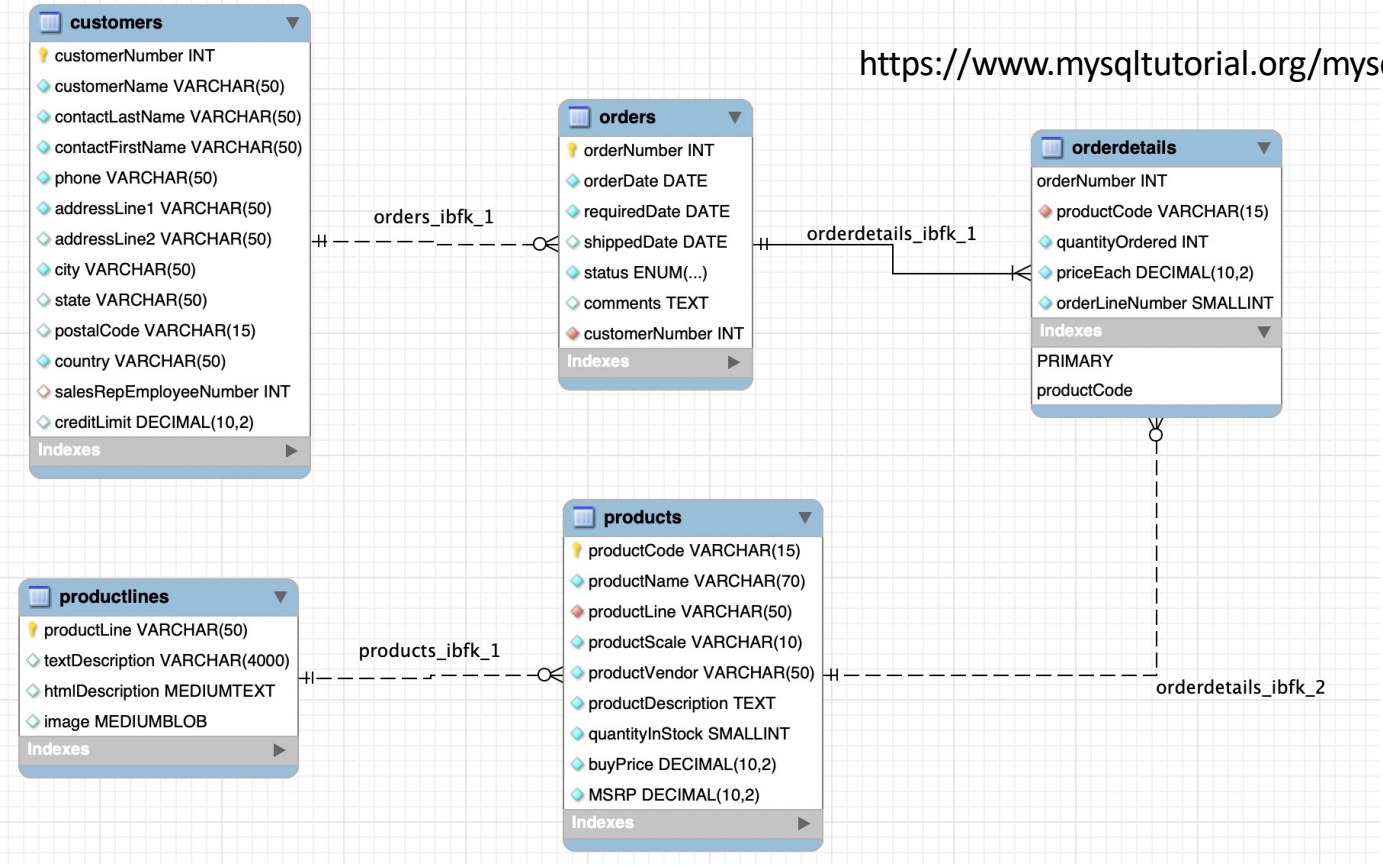
Expressing Weak Entity Sets

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



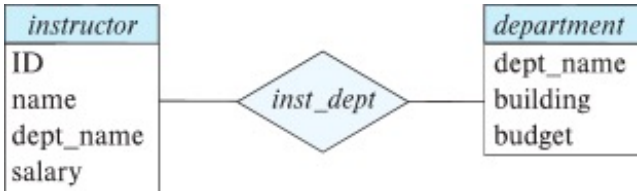
An Example – Classic Models

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



Redundant Attributes

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



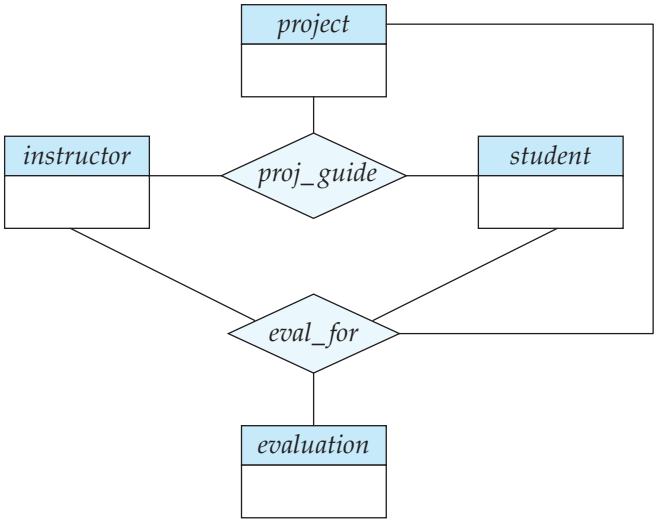
Design Alternatives

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

Emphasis Added

Aggregation

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

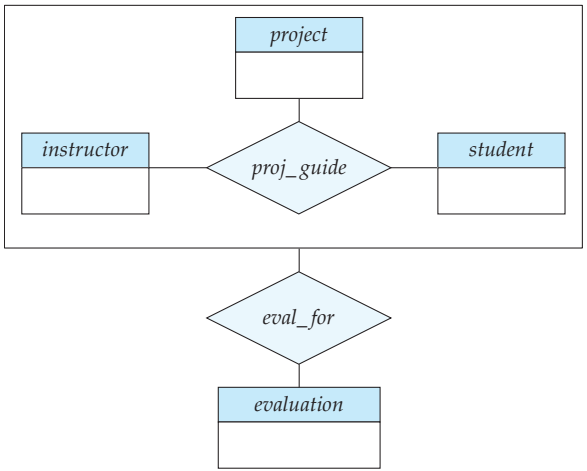


Aggregation (Cont.)

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

Aggregation (Cont.)

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



Reduction to Relational Schemas

- To represent aggregation, create a schema containing
  - Primary key of the aggregated relationship,
  - The primary key of the associated entity set
  - Any descriptive attributes
- In our example:
  - The schema *eval_for* is:

    eval_for (*s_ID, project_id, i_ID, evaluation_id*)
  - The schema *proj_guide* is redundant.

Do Some Worked Examples

Game of Thrones – IMDB
(Show code. Links to IMDB. ... ...)
Columbia Data Model (Bottom Up, Top Down)

Worked Example – Top Down

- Scenario
  - Course with complex course ID.
  - Section
  - Faculty
  - Department
  - Instructor – Department Assoc. entity with properties (role, date).
  - Instructor – Section
  - Student - Section

- Do ER (live) diagram in Lucidchart.

- Do schema creation
  - In DataGrip
  - Show copying statements int the notebook.

Bottom-Up: Some Sources of Information

- Course Codes:
  https://www.cc-seas.columbia.edu/sites/dsa/files/handbooks/Columbia%20Key%20to%20Course%20Listing.pdf
- Course/Section Properties:
  http://www.columbia.edu/cu/bulletin/uwb/
- Common Search Criteria → Indexes, Query Parameters, Data Links
  https://doc.search.columbia.edu/classes/Ferguson?instr=&name=&days=&semes=&hour=&moi=
- Department Information:
  - Script: https://www.columbia.edu/content/academics/departments
  - Codes: https://academic-admin.cuit.columbia.edu/dept_code
- Specific Information:
  - Course and Instructor: https://opendataservice.columbia.edu/course-information
    (They seem to have removed the data. I downloaded two weeks ago).
  - Vergil Data: https://vergil.registrar.columbia.edu/feeds/cw.js
  - Vergil Search: https://vergil.registrar.columbia.edu/doc-adv-queries.php?key=ferguson&moreresults=2

Meet in the Middle

- Show cu_info project with data and processing.
- Course_model schema.
- Show notebook for loading the data.
- Show getting part of the way through parsing HTML in project