
09-13 Lecture 2:

---

cheat sheet

1. Concept: Modeling:

1. Button-up data analysis and modeling: Bottom-up database analysis and modeling is an approach to database design that starts by examining existing data sources, structures, and applications to identify the underlying data requirements. The process gradually builds toward a cohesive database schema by integrating and refining individual data elements and relationships.

1. Identify Existing Data Sources
2. Analyze Existing Data Structures
3. Refine Data Relationships (Normalization)
4. Integrate Data Elements
5. Model the New Schema

After Schema Design:

1. Data Analysis (Warehousing, Dashboards)
2. Operational Data (Interactive CRUD Applications)

2. Top-down data analysis and modeling: An approach to database design that begins by defining the business problem or goal. The top-down approach starts from zero (a high-level idea or problem) and systematically builds up into a complete database.

After defending the problem, it proceeds to create high-level conceptual models, which are then refined into logical and physical models before ultimately being implemented as a database. This method focuses on a structured design process to ensure that the database aligns with the organization's requirements.

1. **Start with the problem** (the "0 point"):
    - Identify what you're trying to achieve, like "I need a system to manage customers and

their orders."

2. **Create a conceptual model** (big-picture planning):
    - Imagine the entities and their relationships, like "A customer places orders, and each order includes products."
3. **Develop a logical model** (add details):
    - Break down the entities into attributes and relationships, like "A customer has a name, email, and phone."
4. **Design a physical model** (ready for implementation):
    - Choose technical details, like which database system to use and how to structure the tables.
5. **Build the database** (implement it):
    - Use tools and code (like SQL) to create the actual database based on your design.

2. Concept: Build Web Application and Development:

- Build a system that create, retrieve, update, delete

1. Resource Model

- **Definition:** A collection of file resources (data entities) in the system.
- **Operations:**
    - **CRUD Operations/ SQL methods**: Create, Retrieve, Update, Delete resources.

2. Server (Backend)

- **2 Components:**
    - **Application Data Model**: Defines how data is structured and organized (e.g., in a database).
    - **Resource Model (API)**: Exposes data and logic to the client via an API (Application Programming Interface).
        - **Transition Property**: Interaction between backend and frontend using APIs.

3. Client (Frontend)

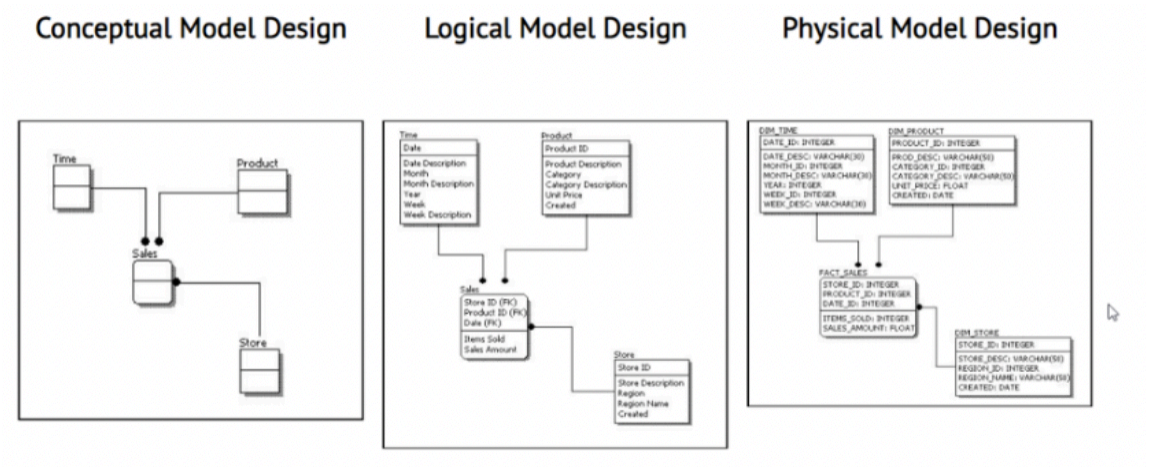
- **Role:** Provides the user interface (UI) for users to interact with the application.
    - **Transition Property:** Communication between the frontend and backend occurs through **HTTP** (HyperText Transfer Protocol).

Four Core Domains of Web Application Development:

1. **Personas, Roles, Goals, and User Stories**
    a. **Personas**: Represent typical users (e.g., Student, Admin).
    b. **Roles**: Different user roles with different privileges (e.g., Admin, User).
    c. **Goals**: What users want to achieve with the app (e.g., Upload a file).
    d. **User Stories**: Scenarios that describe how users interact with the system (e.g., "As a user, I want to upload a file so I can submit it").
2. **User Experience (UX) and User Interface (UI)**
    a. Focus on designing easy-to-use and visually appealing interfaces that allow users to perform actions like uploading, viewing, updating, and deleting resources.
3. **System Design / Component Design / Service Design**
    a. **System Design**: Plan the overall structure of the app, including components like the server, frontend, and database.
    b. **Component Design**: Break down the system into components such as the API, database management, and frontend interface.
    c. **Service Design**: Plan how services (e.g., file storage, user authentication) are integrated into the system.
4. **Data Model/Design**
    a. **Data Model**: Defines the structure of the data, e.g., tables or collections in the database.
    b. **Database Design**: Involves deciding on the schema, relationships between entities (e.g., users, files), and how data is stored and accessed.

Common Approach to Data Modeling:

1. **Conceptual Model:** Focuses on defining the big picture—what entities are involved, how they relate, and what the business goals are.
2. **Logical Model:** Adds structure to the conceptual model, specifying data relationships and defining the schema.
3. **Physical Model:** Translates the logical design into a real-world, implementable system, focusing on storage, performance, and technical considerations.



---

1. Entity-Relation Model

Entities: In an ER model, an entity represents a **real-world object** or concept. Each entity is something about which data is stored. For example:

- **Customer**: An entity representing a customer in a store system.
- **Order**: An entity representing an order placed by a customer.

Attributes: Each entity has **attributes**, which are the properties or characteristics of the entity. For example:

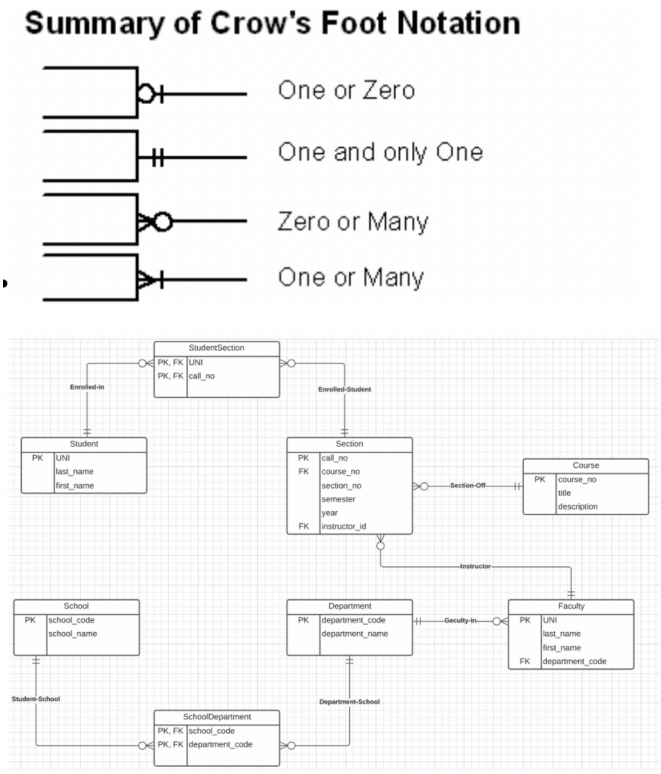
> **Customer** entities might have attributes like `customer_id`, `name`, `email`, etc.

Relationships: Entities are connected by **relationships** that define how they are related to each other. This is detonated by **Crow's Foot Notation**:

Degree of relationship: denotes the number of entities involved in a relationship.

1. Unary relationship: an entity with relation to a itself
2. Binary relationship: two entities share relation with each other
3. Ternary relationship: tree entities involved in a relation

Cardinality: is the relationship between two entities. The constraints (crow's foot notation).



- Entities: student, student sections, department, ect.
- Attributes of student: UNI, last_name, first_name
- Relationship: mapped using the arrows and end point
    - Degree: binary

- **ER Model Advantage**: visualization. Clear relationship between entity and its attributes. Converted into ER model.
- **ER Model Disadvantage**: limited constraints and specification. Loss of content information. No representation of data manipulation. Non industry standard notation.

(1) **Attribute Sets:** An attribute set refers to the collection of attributes associated with an entity in a database. Each attribute represents a specific characteristic or property of the entity.

(2) **Entity sets:** An entity set is a collection of entities that belong to the same schema, and this is why we say that entities in an entity set are of the same type — they share the same schema.

Instructor = (ID, name, salary). Schema is an instructor, and the others are its entities.

(3) **Relationship sets:** A relationship set is called a "set" because it is a collection of relationships, each of which is a tuple representing a specific connection between entities.

A relationship set in the context of Crow's Foot Notation represents the connections between two

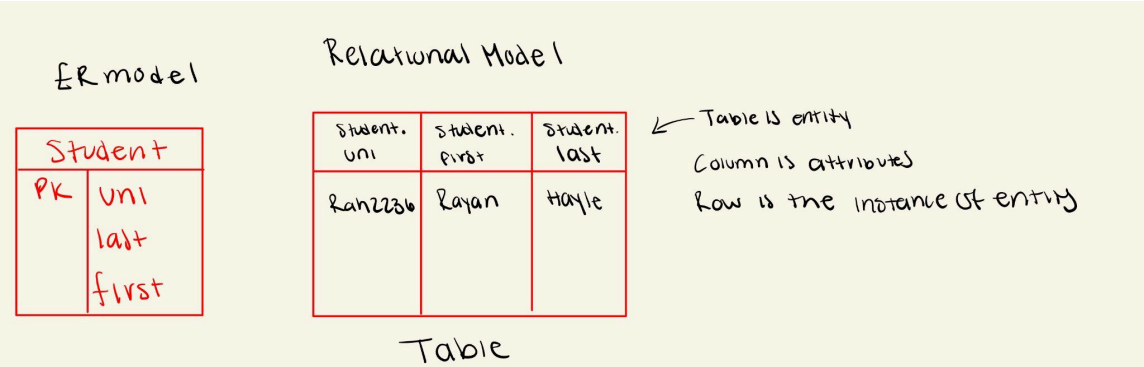
(or more) entities in a database schema.

```
Python
Entity: Students
Attributes: Student_ID, Name, Email

Entity: Courses
Attributes: Course_ID, Course_Name

relationship: Enrolls, many-to-many. (student many course, courses many
students)
the relationship set : many-to-many, many-to-many
```

---

2. Relational Model

- The ER provides the visual representation of the model. And the relational model is the establishment of relations amongst the entities in table form and the base fundamentals of SQL DDL.
- Relation is the table of a schema. The table(entity) has columns (attributes) and each column has data (instance).
- Relations are unordered, meaning the column data is not ordered.
- In table: Entities are the table's title. And the attributes are columns. Each row has data this data is determined by instance of the entity



(1) Attributes

(a) **Domain:** The data type and size stored under the attribute.

- VARCHAR(256), INT(4), NULL, NOT NULL

(b) **Atomic:** attribute values/the instance of entities are normally required to be atomic. This means that the data stored as an instance cannot be further divided.

- Example: SSN → This column can't break down this attribute into multiple pieces
- Example not atomic: Full Name → this attribute/column can be brown down into two more attributes/columns: first name, last name

(c) **Null:** all domains have null value. This means that all columns can have an instance of Null value; this means that the instance is unknown or does not exist.

- When defining a table in SQL; you must state: (1) the entity name, (2) the domain, (3)key, (4)instance can be null or not (primary keys instance can't be null). This is DDL.

(2) Keys

(a) **Primary key:** A subset of attributes in a table that **uniquely identifies** each record (row) in the table. A primary key is a **minimal superkey**.

- STUDENT_ID is primary key, because this column will have only unique rows/data

(b) **Superkey:** (1) Any set of attributes (or columns) that can uniquely identify a row in a table.

(2) A superkey can include the primary key and additional attributes, even though those additional attributes may not be necessary for uniqueness.

- Set of primary keys are superkeys and minimal superkey : {UNI, SNN}
- Primary keys contact with non-unique keys: {name_uni, school_uni, age_SNN}

(c) **Candidate key:** any set of attributes (columns) that can uniquely identify a row in a table, just like the primary key. However, unlike the primary key, a candidate key is **not yet selected** as the main key for the table — it's just a potential key.

- These keys are also minimal sets, meaning that every attribute is unique and no unnecessary attributes are included in the key.
  - Attribute:{ SSN, UNI, Student_ID_Number, name, school, level}
  - Primary key : {UNI}
  - Candidate key: {SSN,Student_ID_Number} each key is a minimal primary key, has potential to be primary key, just not selected

(d) **Composite keys**: A **composite key** is a **primary key** that consists of **two or more**

attributes (columns) that together uniquely identify a row in a table. Each individual attribute in a composite key may **not** uniquely identify a row by itself, but when combined, they form a unique identifier for each row.

- At least one primary attribute to make the set unique
- Attribute:{ SSN, UNI, Student_ID_Number, name, school, level}
- Primary key : {UNI}
- Candidate key: {SSN,Student_ID_Number}
- Compost key: {SSN, name}{ UNI,school} # unique rows, but school,name are not unique alone

(i) **Composite primary key :** set of primary keys to identity a row

- Attribute:{ SSN, UNI, Student_ID_Number, name, school, level}
- Composite primary key: {SSN, UNI}, {SSN, Student_ID_Number} # 6 total cpk

(ii) **Composite Foreign key:** set of primary keys from another table

(e) **Foreign key:** the primary key of an entity being referenced by another entity as its foreign key. This is the only way to establish the Crow's Foot Notation relationship in code.

(i) Referencing

(ii) Referenced

(3) Database Schema

- The logical structure of a database. A database instance is snapshot of the data stored in that schema
    - With keys, entities and attributes we will not establish the DDL
    - Observe: the relations are unordered
        - Relational table name (columns/attributes)

- schema: *instructor* (*ID, name, dept_name, salary*)
- Instance:

, *salary*)

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

(4) Normalization

---

3. Relational Algebra

Relational Algebra Join Operators

- **Select** → σ condition (Table). Conditions → and, or, not → ˆ, ∧, ¬
- **Project** → π attributes/columns (r)
- **Cartesian**→ combine two tables by multiplication → table 1 × Table 2
  - To get the combination of two tables with more control → JOIN operator

1. Inner Joins: returns rows which having matching columns
   a. **Theta Join** 'c' condition → P ⋈ c Q | σ c (P × Q)
   b. **Natural join** Tables based on assumed equal columns → P ⋈ = Q | σ (P × Q)
   c. **Equi Join** equal condition → P ⋈ column = column Q | σ column = column (P × Q)
   d. **Anti Join** Rows in Table X but not Table Y → X ▷ Y | X-X ⋉ Y
   e. **Semi Join:** Rows in X and match to rows in Y
      i. Left: A ⋉ B → All rows in in A that has a match in B, but return only B
      ii. Right: A ⋊ B → All rows in in B that has a match in A, but return only A
2. Outer Join: returns rows/tuples that don't have matching columns. Concat two tables.
   a. Left A ⟕ B → All rows in A, concat to rows in B by on column, if no match on B – NULL.

b. Right A ⟕ B ⟶ All rows in B, match to rows in A by on column, if no match on A- NULL.
c. Full A ⟗ B ⟶ Combine left and right: All rows in A,B, match based on shared column - NULL

- **Query:** refers to the code or expression used to interact with a database to retrieve, manipulate, or manage **data.** 💚
- **Equivalent Queries**: different query equations can produce same result
  - σ dept_name = 'Physics' (instructor ⋈ instructor.ID = teaches.ID teaches)
    - (σ dept_name = 'Physics' (instructor)) ⋈ instructor.ID = teaches.ID teaches
  - σ dept_name = 'Physics' ∧ salary > 90000 (instructor)
    - σ dept_name = 'Physics' (σ salary > 90000 (instructor))

Python

```
# 1. SELECT OPERATION

σ dept_name = 'Biology' (department) # returns rows where dept_name = Biology

σ dept_name = 'Comp. Sci' ∧ salary > 10000 (instructor)


# 2. PROJECT OPERATION

π ID,name (instructor)

π building,dept_name,budget (σ building = 'Taylor' and dept_name = 'Comp. Sci'
and budget = 100000(department))

π ID,name (instructor) # find name of all instructors who teach biology

# 3. CARTESIAN OPERATION

instructor × department # 12 * 7 = 84 new rows
```

Python

```
# Joins: Inner
```

```
Python
# Joins: Outter
```

SQL Join Operators

- ON during the join, WHERE after the join

(1) Select, Project → SELECT <columns> FROM <table> WHERE <condition>

(2) Cartesian Product→ SELECT <column_1, column_2> FROM <table_1> CROSS JOIN <table_2 > ON <condition>

→ SELECT <column_1, column_2> FROM <table_1> CROSS JOIN <table_2 > ;

(3) Natural Join → SELECT <column_1, column_2> FROM <table_1> NATURAL JOIN <table_2 > WHERE <condition>

(4) Inner Join → SELECT <column_1, column_2> FROM <table_1> INNER JOIN <table_2 > ON <condition>

(5) Right/Left Join → SELECT <column_1, column_2> FROM <table_1> LEFT/RIGHT JOIN <table_2 > ON <condition>

(6) Full Outer Join → SELECT <column_1, column_2> FROM <table_1> FULL OUTER JOIN <table_2 > ON <condition>

```
Python
# Cartesian product

select * from classroom, department

# 1. Equi join

select name, course_id

from instructor,teaches

where instructor.ID = teaches.ID and instructor.dept_name = 'Art'

# 2. Natural Join

select name, course_id

from student NATURAL JOIN takes;

where takes.course_id = course.course_id;
```

```
# Natural join Using clause - be specific about the columns

select name, title

from (student natural join takes) join course USING (course_id)
```

Python

```
# Inner and Outer Joins
```

---

4. SQL

Core Methods:

- Select returns columns like project does

(1) Remove Duplicates → SELECT DISTINCT <columns> FROM <table> WHERE <condition>

(2) Keep Duplicates → SELECT ALL <columns> FROM <table> WHERE <condition>

(3) Select and arithmetic +,-,*,/ → SELECT ALL <columns> * operation FROM <table> WHERE <condition>
→ rename the operator table using AS

Python

```
# 1. return all columns
select * from department
# 2. only unique building column return
select distinct building from  classroom

# 3. return all duplicates of this column
SELECT ALL dept_name from instructor

# 4. select +  arithmetic operators
SELECT ID, name, salary/12 AS monthy_salary
FROM instructor

select building,room_number,capacity + 3 as NEW_CAP
FROM classroom
```

4A. SQL Data Definition Language (DDL)

(1) Defining the schema of database

```sql
-- Creating the Students table
CREATE TABLE Students (
    StudentID INT NOT NULL,           -- Attribute: StudentID, Domain: INT, Instance: UNIQUE, NOT NULL
    FirstName VARCHAR(50),            -- Attribute: FirstName, Domain: VARCHAR(50), Instance: Can be NULL
    LastName VARCHAR(50),             -- Attribute: LastName, Domain: VARCHAR(50), Instance: Can be NULL
    PRIMARY KEY (StudentID)           -- Primary Key: StudentID (this uniquely identifies each student)
);

-- Creating the Courses table
CREATE TABLE Courses (
    CourseID INT NOT NULL,            -- Attribute: CourseID, Domain: INT, Instance: UNIQUE, NOT NULL
    CourseName VARCHAR(100),          -- Attribute: CourseName, Domain: VARCHAR(100), Instance: Can be NULL
    PRIMARY KEY (CourseID)            -- Primary Key: CourseID (this uniquely identifies each course)
);

-- Creating the Enrollments table
CREATE TABLE Enrollments (
    EnrollmentID INT NOT NULL,        -- Attribute: EnrollmentID, Domain: INT, Instance: UNIQUE, NOT NULL
    StudentID INT,                    -- Attribute: StudentID, Domain: INT, Instance: Can be NULL (foreign key)
    CourseID INT,                     -- Attribute: CourseID, Domain: INT, Instance: Can be NULL (foreign key)
    PRIMARY KEY (EnrollmentID),       -- Primary Key: EnrollmentID (this uniquely identifies each enrollment)
    FOREIGN KEY (StudentID) REFERENCES Students(StudentID),  -- Foreign Key: StudentID references Students table
    FOREIGN KEY (CourseID) REFERENCES Courses(CourseID)      -- Foreign Key: CourseID references Courses table
);
```

(2) Rename operation → Old_name AS New_name

```python
# Rename
# instructors whose salary is higher than cs instructors
select distinct not_cs.name
from instructor as not_cs, instructor as is_cst
where not_cs.salary > is_cst.salary and is_cst.dept_name = 'Comp. Sci.'
```

4B. SQL Data Modification Language (DML)

(1) Deletion

- Delete the entire table → DELETE FROM <table>
- Delete all → DELETE FROM <table> WHERE <condition>

```python
# delete all
delete from instructor

# delete all instructor from fiance
delete from instructor
where dept_name = 'Finace';

# delete all instructor: if department is in watson

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

# delete instructor if their salary is lower than the avg
delete from instructor
where salary < (select avg(salary) from instructor);
```

(2) Insertion

- INSERT INTO <table> VALUES (); | INSERT INTO <table> < columns> VALUES ()
- INSERT INTO <table>

```python
# Add new course
INSERT INTO course
values('CS-437', 'Database','Comp.Sci', 4);

INSERT INTO course (course_id, title, dept_name,credits)
values('CS-437', 'Database','Comp.Sci', 4);



# All students who earned more 114 in music --> make then instructor, salary =
1800
INSERT into instructor
select ID, name, dept_name, 1800
from student
where dept_name = 'Music' and tot_cred > 144;
```

(3) Update

- UPDATE *table_name*

- `SET column1 = value1, column2 = value2, ...`
- `WHERE condition;`

```python
# 5% raise to all instructors

update instructor
set salary = instructor * 1.05

# only for teachers who earn less than 70,500
update instructor
set salary = salary * 1.5
where salary < 70500;

# only for those who earn less than the avg
update instructor
set salary = salary * 1.5
where salary < (select avg(salary) from instructor)

# two updates once over 100k raise by 3, else 5%
update instructor
       set salary = salary * 1.03
       where salary > 100000
update instructor
       set salARY = Salary * 1.05
       where salary <=100000
```

09-20 Lecture 3:

---

1. Entity-Relation Model

(1) Associative entity

(2) Relationship sets with attributes

(3) Mapping Cardinality Constraints

4A. SQL Data Definition Language (DDL)

(1) **Integrity Constraints:** guard against accidental damage to the database.

(a) Candidate keys can be Null, primary keys cannot be Null

(2) **Unique Constraints:** All values of candidate keys must be unique

```
Unset
CREATE TABLE Enrollments (
    EnrollmentID INT NOT NULL,      -- Primary key for the enrollment record
    StudentID INT,                  -- Candidate key, unique in combination
with CourseID
    CourseID INT,                   -- Course the student is enrolled in
    PRIMARY KEY (EnrollmentID),     -- Ensures each enrollment has a unique ID
    UNIQUE (StudentID, CourseID)    -- Ensures a student cannot enroll in the
same course twice
);
```

(3) **Check Clause:** specific predicate P that must be satisfied by every row

- CHECK (attribute IN <condition>)
- CONSTRAINT constraint_name CHECK (condition)

```
Unset
CREATE TABLE section(

course_id VARCHAR(8),
sec_id VARCHAR(8),
semester VARCHAR(6),
year NUMERIC(4,0),
room_number VARCHAR(7),
time_Slot VARCHAR(8),

PRIMARY KEY (course_id,sec_id,semester,year)

# make sure every row has semester and data is 4 options

CHECK(sumter IN ('Fall','Winter','Spring','Summer'))
)
```

Referential Integrity

- Referential integrity ensures that relationships between tables in a relational database remain consistent. If a value appears in one relation (table) for a given set of attributes, it must also appear in the corresponding attributes in another relation.
    - If we have two relations, say **R** and **S**, and an attribute **A** is the **primary key** of relation **S**, then by the rules of referential integrity, **A** must also be a **foreign key** in relation **R**.

        - This relationship and consistency is maintained through foreign keys
            - Basically: When a table has foreign keys is referencing another table and its primary key

Cascading Actions In Referential Integrity

- When the referential integrity is violated, the normal produce will reject the actions that caused the violation
- **Cascading** is a **proactive mechanism** in the database that **automatically handles referential integrity violations** caused by changes to the referenced table. Instead of rejecting the action or requiring you to manually handle the changes in the referencing table, cascading ensures that changes propagate consistently across related tables.
    - When a referenced row (a primary key) is **deleted**, the corresponding rows in the referencing table (foreign key) are **automatically deleted**.
    - you must handle the referencing table **manually** before making changes to the referenced table. However, if you define cascading actions, the database will handle these dependencies for you.
        - ON UPDATE CASCADE, ON DELETE CASCADE
        - ON UPDATE SET DEFAULT/NULL, ON DELETE SET DEFAULT/NULL
            - Both of these two options handle the violation error

```
Python
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY
);

CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);

-- Insert data
INSERT INTO Customers (CustomerID) VALUES (1);
INSERT INTO Orders (OrderID, CustomerID) VALUES (101, 1);

#-- Attempt to delete CustomerID = 1
DELETE FROM Customers WHERE CustomerID = 1;
```

```
-- ERROR: Referential integrity violation because Orders references CustomerID
= 1
```

```
Python
# use concasing to handle the error the reference will cause

CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY
);

CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
             ON DELETE CASCADE
             on UPDATE CASCADE
);

#-- Insert data
INSERT INTO Customers (CustomerID) VALUES (1);
INSERT INTO Orders (OrderID, CustomerID) VALUES (101, 1);

#-- Attempt to delete CustomerID = 1
DELETE FROM Customers WHERE CustomerID = 1;
#-- NO Error
```

09-27 Lecture 4:

CODD'S 12 RULES → What's considered A Database

Rule 1: Information Rule → All data is stored in tables, and everything is represented as rows and columns.

Rule 2: Guaranteed Access Rule →Every value in the database can be accessed by specifying its table, column, and primary key.

Rule 3: Systematic Treatment of NULL Values ⟶ NULL values are uniformly treated to represent missing or inapplicable data, without ambiguity.

Rule 4: Dynamic Online Catalog → Metadata (data about data) is stored in the database and accessible via standard SQL queries.

Rule 5: Comprehensive Data Sub-Language → The database must support a language (like SQL) for data definition, manipulation, and querying.

Rule 6: View Updating Rule →Views (virtual tables) must be updatable when theoretically possible.

Rule 7: High-Level Insert, Update, and Delete → Operations like insertion, deletion, and updates must be supported for sets of data, not just single rows.

Rule 8: Physical Data Independence →Changes in physical storage do not affect the logical structure or how data is accessed.

Rule 9: Logical Data Independence → Changes to the logical structure of the database (e.g., adding a column) do not affect existing applications.

Rule 10: Integrity Independence → Integrity constraints (e.g., primary keys, foreign keys) must be definable and stored within the database.

Rule 11: Distribution Independence → The database should work the same whether it is distributed across multiple locations or centralized.

Rule 12: Non-Subversion Rule → All operations, even those at a lower level, must adhere to the rules of relational integrity.

4. SQL

Null Values

- In RA you can check null using select →σ prereq_id = NULL (course ⋈ prereq_id = course_id prereq)
- In SQL using IS methods to check if something is Null not =

Aggregate Function

- An aggregate function : Predefined functions in SQL
    - Avg, min, max, sum, count
    - This is a two step process.
- **Where** clause happens before the grouping, **having** happens to the after, to the result of the grouping

```Python
# the aggregate functions - avg, sum ,min, max, count

# 1. avg(column), min(column), max (column)
select avg(salary)
from instructor
where dept_name = 'Biology';
```

```
select min(ID) from instructor;

# 2. group by
select max(salary)
from instructor
group by dept_name; # only returns the max(salary) and nothing showing what dept it is

select dept_name, avg(salary) as the_avg
from instructor
group by dept_name; # if it's groupby biology you can't project columns that will be lost by the grouping like name,id

# 3. COUNT
# find total of teachers, teaching spring 2018
select  count(ID)
from teaches
where year = 2018 and semester = 'Spring'; # only the count

select *
from  teaches
where year = 2018 and semester = 'Spring'; # get all of their info

select count(*)
from teaches; # count all rows in teaches, all of teachers

# Having (condition) - this is after the group by. Where is before the groupby


# find the richest in each dept, only return if they earn more than 90k

select dept_name, max(salary)
from instructor
group by dept_name
having max(salary) > 90000;
```

4C. SQL Joins

-

(1) SQL operations

(2) Nested Subqueries

(3) Set membership

(4) Set comparison

  (a) Some Clause

  (b) All Clause

(5) Empty Relations
(6) Exists Clause
(7) Not Exists Clause
(8) Test Absence Duplicate Tuples
(9) Subqueries in the from Clause
    (a) With Clause
(10) Scalar Subquery

10-04 Lecture 5:

---

1. Entity-Relation Model:

- In sql and strict relational the only way to relate two entities is to use associative entities or foreign keys. So that you can compute them with joins
- When people describe relationships with verbs. These are natural, obvious relationships.
    - Student (entity) enrolls(relation) in class(entity) for foreign keys, associate entities
- Other kind of relationship that is not easily represented
    - This relationship is ISA
        - A student isa person, professor isa person
- A relation is an entity set, but when you do an ISA relationship, how do you model this entity?
    - ISA is diamond shaped

(1) Inheritance:

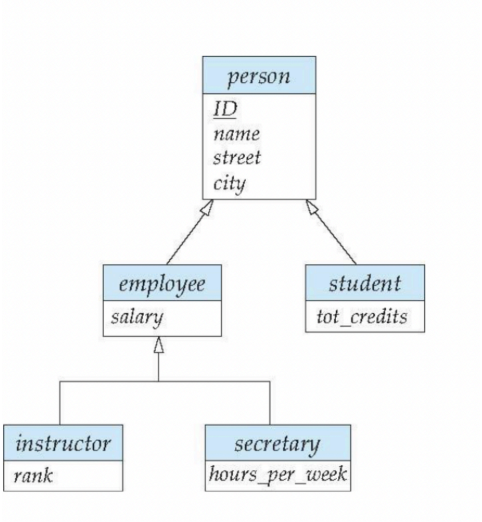
- OPP : child class to **inherit** the properties (variables) and behaviors (methods) of parent/another class.

  Class: person, has properties x.

  Class: instructor inherits properties x from Person

- The two term specialization and generalization

(a) Specialization



- Person → two specialization employee and student
    - Employee has all the attributes of Person
    - Instructor, secretary are employees. They have attributes of employee and employee has attributes of person
- Overlapping: employee and student. The → are overlapping
- Disjoint: instructor and secretary. The both lines are disjoin
- Total: It means **every instance of the parent entity** must be a member of at least **one child entity** (subclass)
    - Every person is at least employee or student, or both
- Partial: when parent entity is not a member of any of its child entities
    - person that's neither employee or student