



# Chapter 1: Introduction

Database System Concepts, 7<sup>th</sup> Ed.

©Silberschatz, Korth and Sudarshan

See [www.db-book.com](http://www.db-book.com) for conditions on re-use



# Database Applications Examples

- Enterprise Information
  - Sales: customers, products, purchases
  - Accounting: payments, receipts, assets
  - Human Resources: Information about employees, salaries, payroll taxes.
- Manufacturing: management of production, inventory, orders, supply chain.
- Banking and finance
  - customer information, accounts, loans, and banking transactions.
  - Credit card transactions
  - Finance: sales and purchases of financial instruments (e.g., stocks and bonds; storing real-time market data)
- Universities: registration, grades



# Database Applications Examples (Cont.)

- Airlines: reservations, schedules
- Telecommunication: records of calls, texts, and data usage, generating monthly bills, maintaining balances on prepaid calling cards
- Web-based services
  - Online retailers: order tracking, customized recommendations
  - Online advertisements
- Document databases
- Navigation systems: For maintaining the locations of various places of interest along with the exact routes of roads, train systems, buses, etc.



# Purpose of Database Systems

In the early days, database applications were built directly on top of file systems, which leads to:

- Data redundancy and inconsistency: data is stored in multiple file formats resulting in duplication of information in different files
- Difficulty in accessing data
  - Need to write a new program to carry out each new task
- Data isolation
  - Multiple files and formats
- Integrity problems
  - Integrity constraints (e.g., account balance > 0) become “buried” in program code rather than being stated explicitly
  - Hard to add new constraints or change existing ones



# Purpose of Database Systems (Cont.)

- Atomicity of updates
  - Failures may leave database in an inconsistent state with partial updates carried out
  - Example: Transfer of funds from one account to another should either complete or not happen at all
- Concurrent access by multiple users
  - Concurrent access needed for performance
  - Uncontrolled concurrent accesses can lead to inconsistencies
    - Ex: Two people reading a balance (say 100) and updating it by withdrawing money (say 50 each) at the same time
- Security problems
  - Hard to provide user access to some, but not all, data

**Database systems offer solutions to all the above problems**



# Data Models

- A collection of tools for describing
  - Data
  - Data relationships
  - Data semantics
  - Data constraints
- Relational model
- Entity-Relationship data model (mainly for database design)
- Object-based data models (Object-oriented and Object-relational)
- Semi-structured data model (XML)
- Other older models:
  - Network model
  - Hierarchical model



# Relational Model

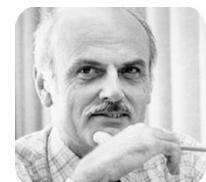
- All the data is stored in various tables.
- Example of tabular data in the relational model

Columns

Rows

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

(a) The *instructor* table



**Ted Codd**  
Turing Award 1981



# A Sample Relational Database

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

(a) The *instructor* table

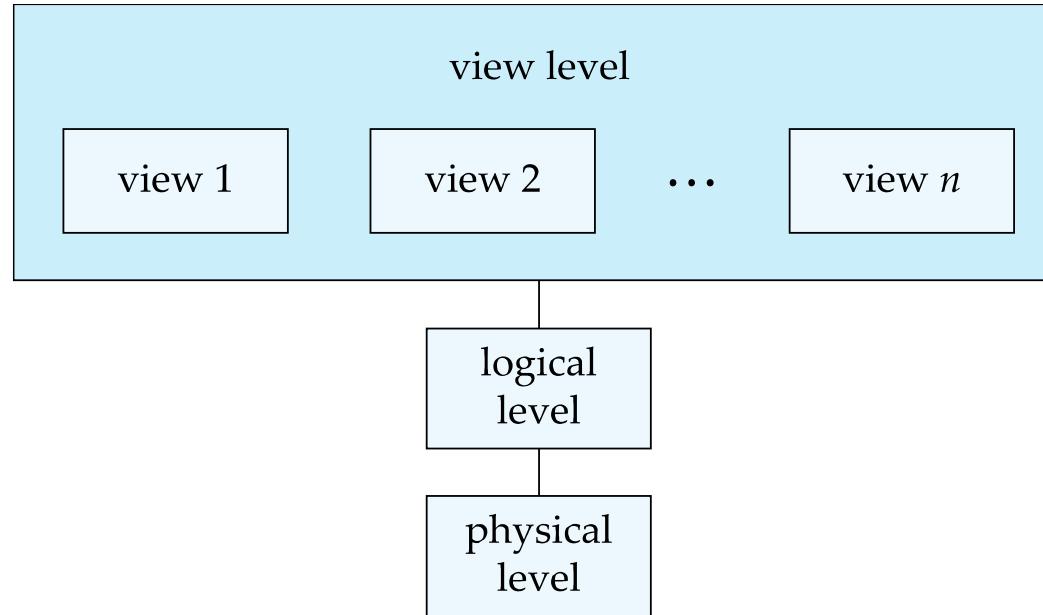
<i>dept_name</i>	<i>building</i>	<i>budget</i>
Comp. Sci.	Taylor	100000
Biology	Watson	90000
Elec. Eng.	Taylor	85000
Music	Packard	80000
Finance	Painter	120000
History	Painter	50000
Physics	Watson	70000

(b) The *department* table



# View of Data

An architecture for a database system





# Instances and Schemas

- Similar to types and variables in programming languages
- **Logical Schema** – the overall logical structure of the database
  - Example: The database consists of information about a set of customers and accounts in a bank and the relationship between them
    - Analogous to type information of a variable in a program
- **Physical schema** – the overall physical structure of the database
- **Instance** – the actual content of the database at a particular point in time
  - Analogous to the value of a variable



# 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 Manipulation Language (DML)

- Language for accessing and updating the data organized by the appropriate data model
  - DML also known as query language
- There are basically two types of data-manipulation language
  - **Procedural DML** -- require a user to specify what data are needed and how to get those data.
  - **Declarative DML** -- require a user to specify what data are needed without specifying how to get those data.
- Declarative DMLs are usually easier to learn and use than are procedural DMLs.
- Declarative DMLs are also referred to as non-procedural DMLs
- The portion of a DML that involves information retrieval is called a **query** language.



# SQL Query Language

- SQL query language is nonprocedural. A query takes as input several tables (possibly only one) and always returns a single table.
- Example to find all instructors in Comp. Sci. dept

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

- SQL is **NOT** a Turing machine equivalent language
- To be able to compute complex functions SQL is usually embedded in some higher-level language
- Application programs generally access databases through one of
  - Language extensions to allow embedded SQL
  - Application program interface (e.g., ODBC/JDBC) which allow SQL queries to be sent to a database



# Database Access from Application Program

- Non-procedural query languages such as SQL are not as powerful as a universal Turing machine.
- SQL does not support actions such as input from users, output to displays, or communication over the network.
- Such computations and actions must be written in a **host language**, such as C/C++, Java or Python, with embedded SQL queries that access the data in the database.
- **Application programs** -- are programs that are used to interact with the database in this fashion.



# Database Design

The process of designing the general structure of the database:

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



# Transaction Management

- A **transaction** is a collection of operations that performs a single logical function in a database application
- **Transaction-management component** ensures that the database remains in a consistent (correct) state despite system failures (e.g., power failures and operating system crashes) and transaction failures.
- **Concurrency-control manager** controls the interaction among the concurrent transactions, to ensure the consistency of the database.



# Database Architecture

- Centralized databases
  - One to a few cores, shared memory
- Client-server,
  - One server machine executes work on behalf of multiple client machines.
- Parallel databases
  - Many core shared memory
  - Shared disk
  - Shared nothing
- Distributed databases
  - Geographical distribution
  - Schema/data heterogeneity



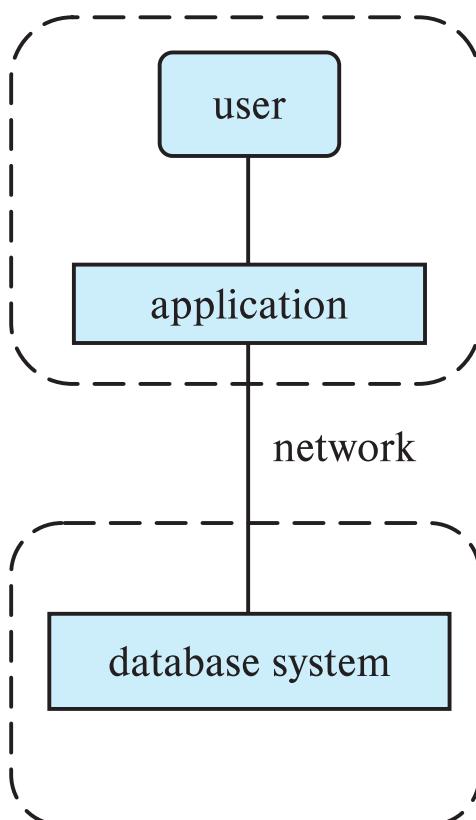
# Database Applications

Database applications are usually partitioned into two or three parts

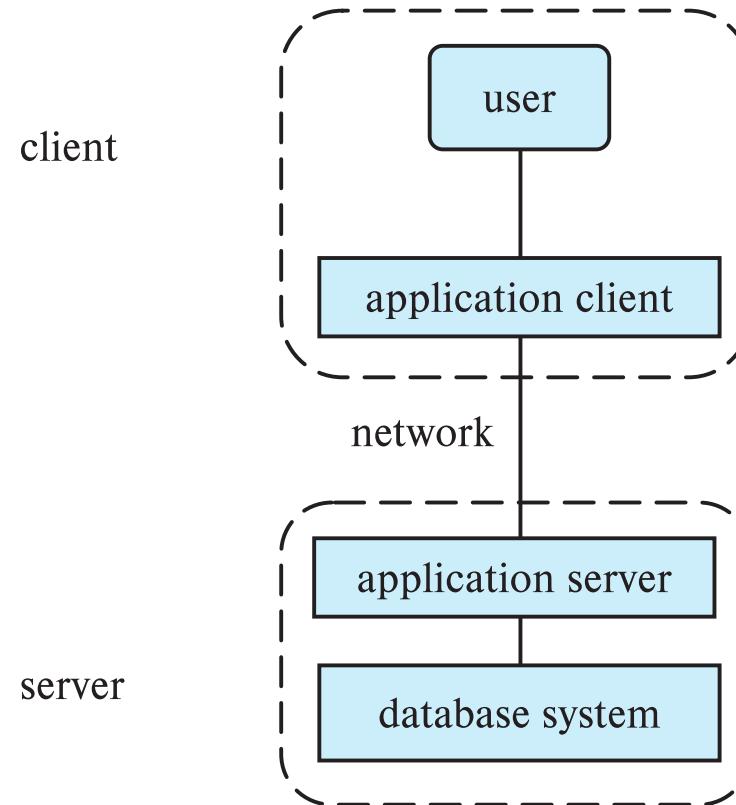
- Two-tier architecture -- the application resides at the client machine, where it invokes database system functionality at the server machine
- Three-tier architecture -- the client machine acts as a front end and does not contain any direct database calls.
  - The client end communicates with an application server, usually through a forms interface.
  - The application server in turn communicates with a database system to access data.



# Two-tier and three-tier architectures



(a) Two-tier architecture



(b) Three-tier architecture



# Database Administrator

A person who has central control over the system is called a **database administrator (DBA)**. Functions of a DBA include:

- Schema definition
- Storage structure and access-method definition
- Schema and physical-organization modification
- Granting of authorization for data access
- Routine maintenance
- Periodically backing up the database
- Ensuring that enough free disk space is available for normal operations, and upgrading disk space as required
- Monitoring jobs running on the database



# History of Database Systems

- 1950s and early 1960s:
  - Data processing using magnetic tapes for storage
    - Tapes provided only sequential access
  - Punched cards for input
- Late 1960s and 1970s:
  - Hard disks allowed direct access to data
  - Network and hierarchical data models in widespread use
  - Ted Codd defines the relational data model
    - Would win the ACM Turing Award for this work
    - IBM Research begins System R prototype
    - UC Berkeley (Michael Stonebraker) begins Ingres prototype
    - Oracle releases first commercial relational database
  - High-performance (for the era) transaction processing



# History of Database Systems (Cont.)

- 1980s:
  - Research relational prototypes evolve into commercial systems
    - SQL becomes industrial standard
  - Parallel and distributed database systems
    - Wisconsin, IBM, Teradata
  - Object-oriented database systems
- 1990s:
  - Large decision support and data-mining applications
  - Large multi-terabyte data warehouses
  - Emergence of Web commerce



# History of Database Systems (Cont.)

- 2000s
  - Big data storage systems
    - Google BigTable, Yahoo PNuts, Amazon,
    - “NoSQL” systems.
  - Big data analysis: beyond SQL
    - Map reduce and friends
- 2010s
  - SQL reloaded
    - SQL front end to Map Reduce systems
    - Massively parallel database systems
    - Multi-core main-memory databases



# End of Chapter 1



# Chapter 6: Database Design Using the E-R Model

Database System Concepts, 7<sup>th</sup> Ed.

©Silberschatz, Korth and Sudarshan

See [www.db-book.com](http://www.db-book.com) for conditions on re-use



# Design Phases

- **Initial phase**

characterize fully the data needs of the prospective database users.

- Second phase (Conceptual)

- Applying the concepts of the chosen data model.
- Translating these requirements into a conceptual schema of the database (names of entities and relationships).
- A fully developed conceptual schema indicates the functional requirements of the enterprise.
  - Describe the kinds of operations (or transactions) that will be performed on the data.



# Design Phases (Cont.)

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



# Design Approaches

- **Entity Relationship Model (covered in this chapter)**
  - Models an enterprise as a collection of *entities* and *relationships*
    - Entity: a “thing” or “object” in the enterprise that is distinguishable from other objects
      - Described by a set of *attributes*
    - Relationship: an association among several entities
      - Represented diagrammatically by an *entity-relationship diagram*:
- **Normalization Theory (Chapter 7)**
  - Formalize what designs are bad, and test for them



# Outline of the ER Model



# Entity Sets

- A database can be modeled as a collection of entities and relationships among them.
- An **entity** is an object that exists and is distinguishable from other objects.
  - Example: specific person, company, event, plant
- An **entity set** is a set of entities of the same type that share the same properties.
  - Example: set of all persons, companies, trees, holidays
- An **entity** is represented by a set of attributes; i.e., descriptive properties possessed by all members of an entity set.
  - Example:
    - $\text{instructor} = (\text{ID}, \text{name}, \text{salary})$
    - $\text{course} = (\text{course\_id}, \text{title}, \text{credits})$



# Representing Entity sets in ER Diagram

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

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

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



# Relationship Sets

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

Example:

44553 (Peltier)  
*student entity*

advisor  
relationship set

22222 (Einstein)  
*instructor entity*

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

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

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

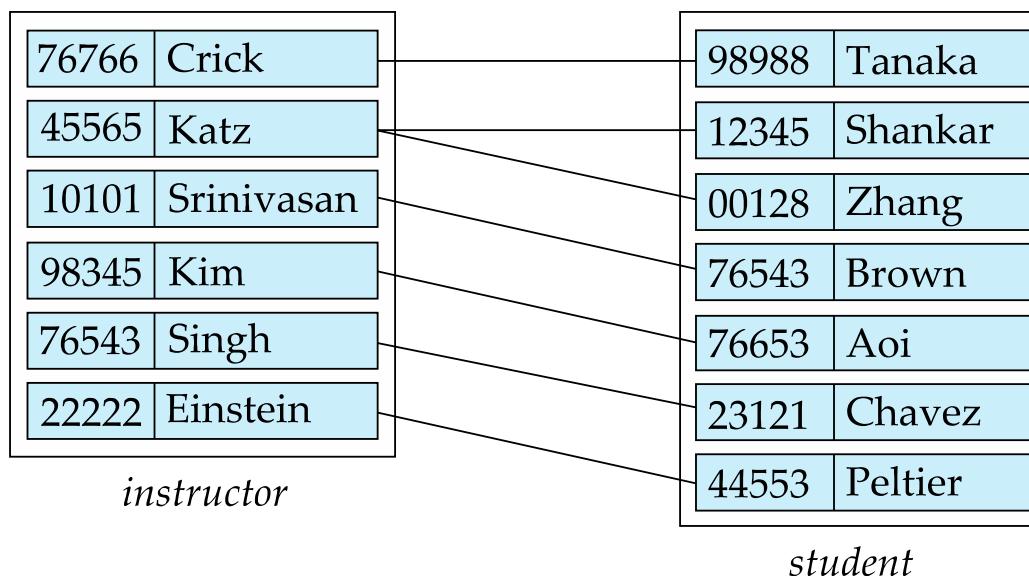
- Example:

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



# Relationship Sets (Cont.)

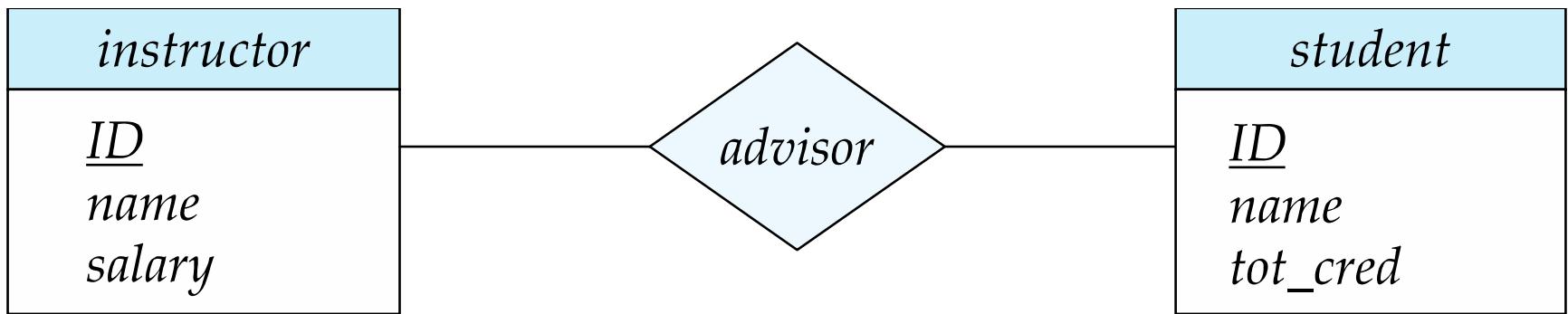
- Example: we define the relationship set *advisor* to denote the associations between students and the instructors who act as their advisors.
- Pictorially, we draw a line between related entities.





# Representing Relationship Sets via ER Diagrams

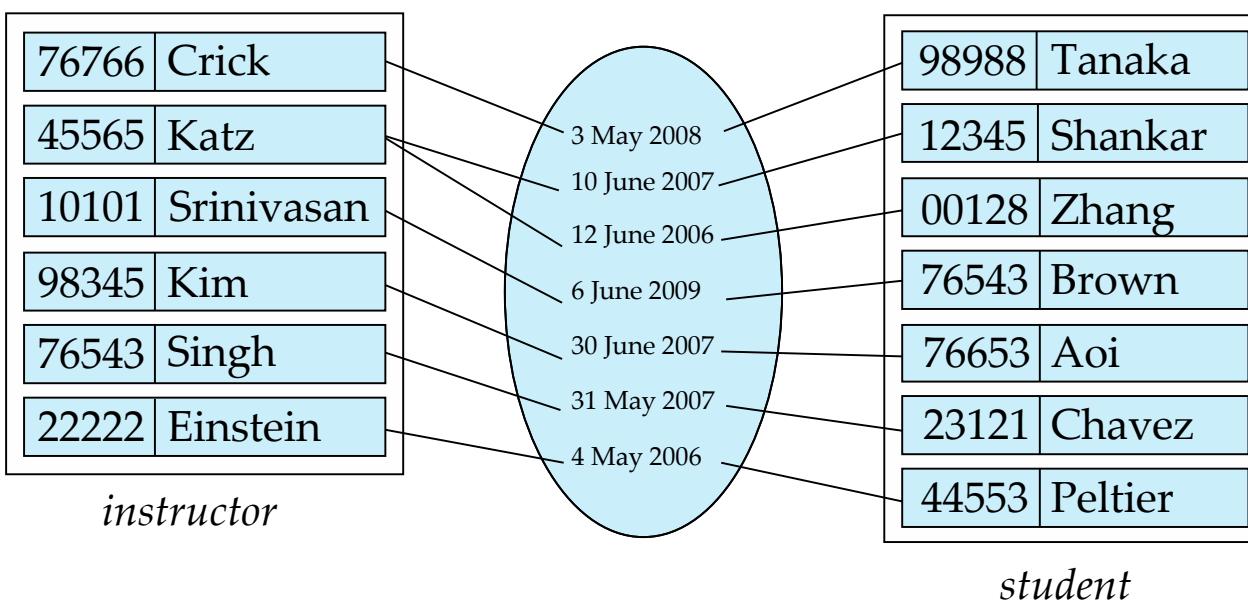
- Diamonds represent relationship sets.





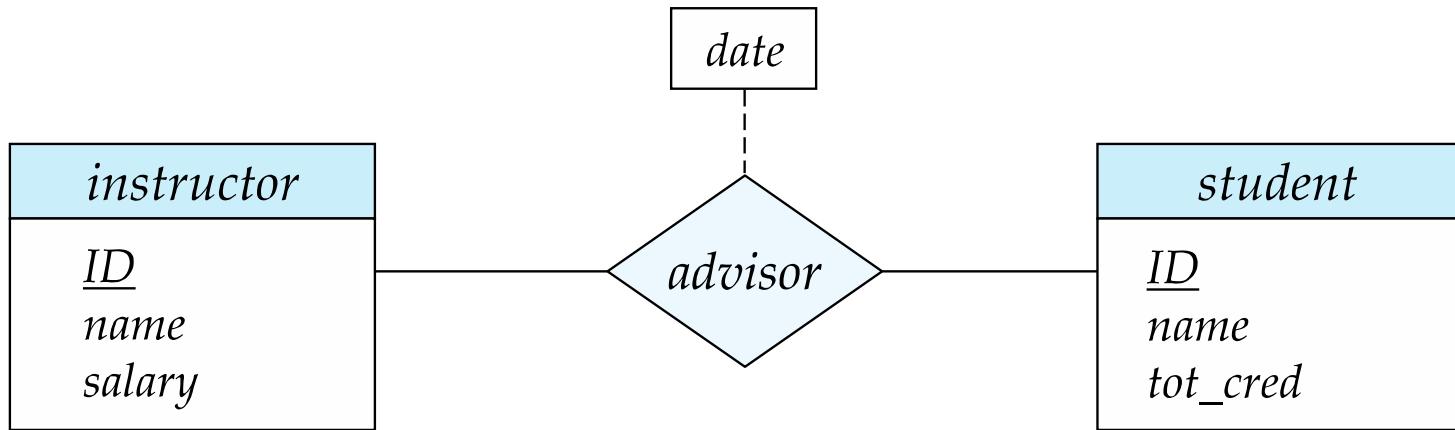
# Relationship Sets (Cont.)

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





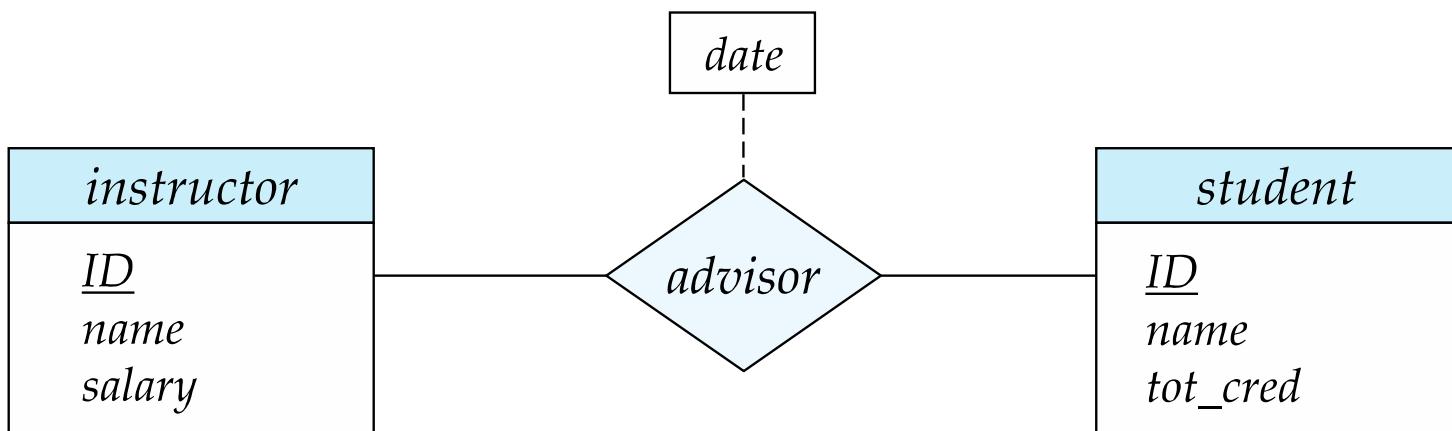
# Relationship Sets with Attributes





# Degree of a Relationship Set

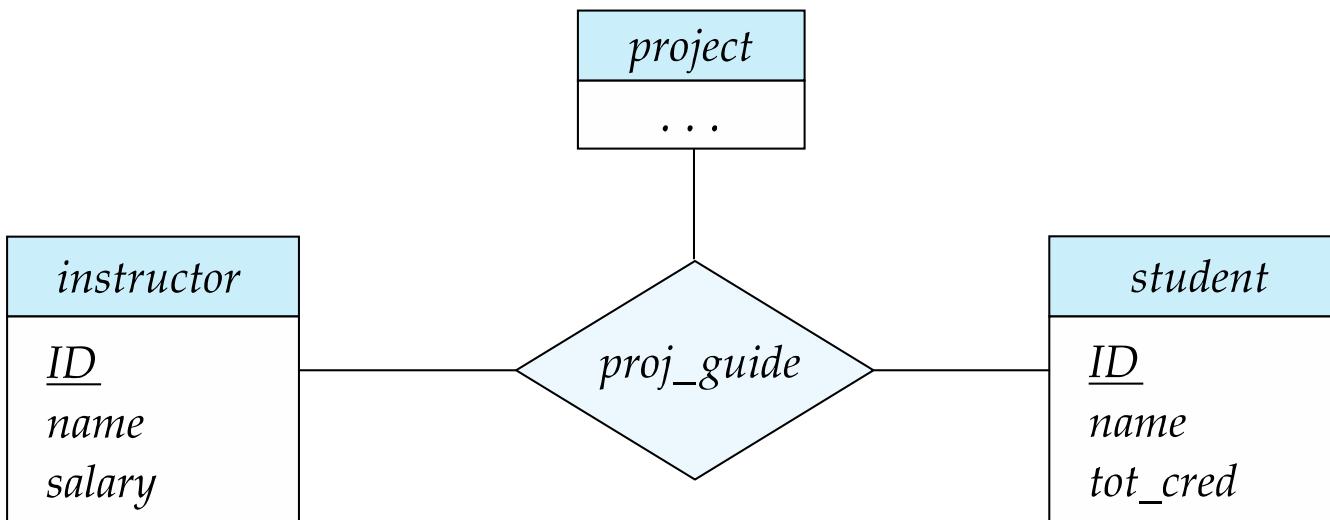
- Binary relationship
  - involve two entity sets (or degree two).
  - most relationship sets in a database system are binary.





# Non-binary Relationship Sets

- Relationships between more than two entity sets are rare. Most relationships are binary. (More on this later.)
  - Example: *students* work on research *projects* under the guidance of an *instructor*.
  - relationship *proj\_guide* is a ternary relationship between *instructor*, *student*, and *project*





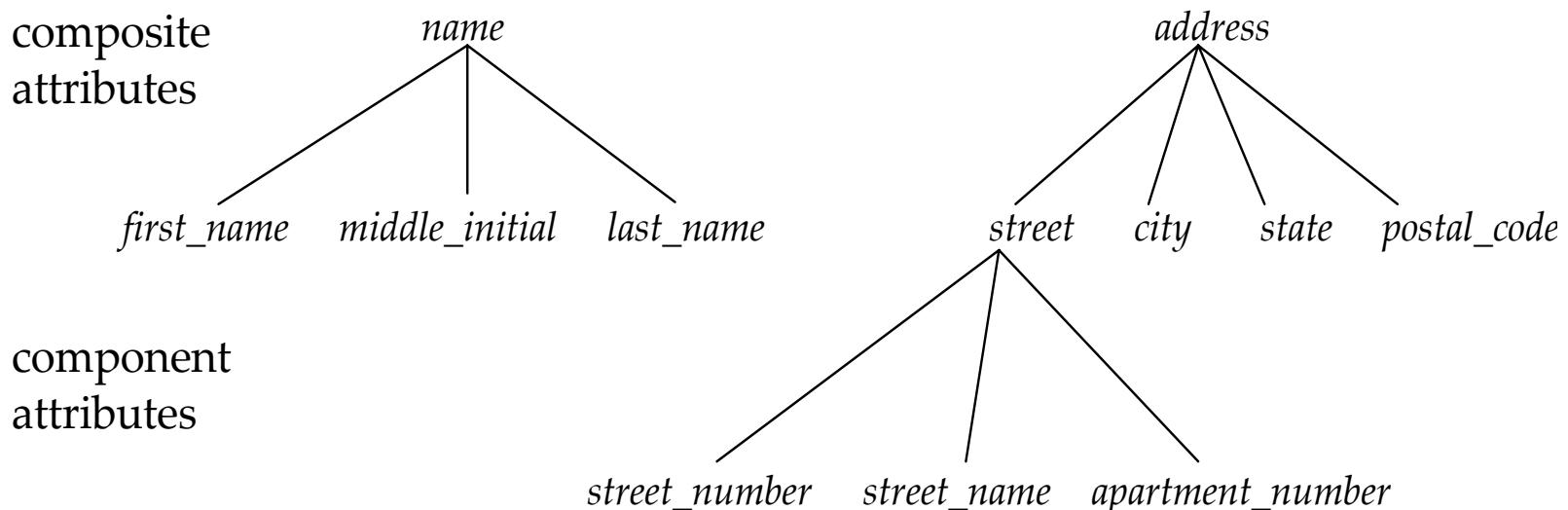
# Attributes

- An **entity** is represented by a set of attributes, that is descriptive properties possessed by all members of an entity set.  
Examples: instructor = (ID, name, street, city, salary )  
course= (course\_id, title, credits)
- Attribute types:
  - **Simple** and **composite** attributes.
  - **Single-valued** and **multivalued** attributes
    - Example: multivalued attribute: *phone\_numbers*
  - **Derived** attributes
    - Can be computed from other attributes
    - Example: age, given date\_of\_birth
- **Domain** – the set of permitted values for each attribute



# Composite Attributes

- Composite attributes allow us to divide attributes into subparts (other attributes).





# Representing Complex Attributes in ER Diagram

<i>instructor</i>	
<u>ID</u>	
<i>name</i>	
<i>first_name</i>	
<i>middle_initial</i>	
<i>last_name</i>	
<i>address</i>	
<i>street</i>	
<i>street_number</i>	
<i>street_name</i>	
<i>apt_number</i>	
<i>city</i>	
<i>state</i>	
<i>zip</i>	
{ <i>phone_number</i> }	
<i>date_of_birth</i>	
<i>age ()</i>	

Let us  
draw it



# Primary Key

- Primary keys provide a way to specify how entities and relations are distinguished. We will consider:
  - Entity sets
  - Relationship sets.
  - Weak entity sets.



# Primary key for Entity Sets

A **super key** of an entity set is a set of one or more attributes whose values uniquely determine each entity.

A **candidate key** of an entity set is a minimal super key

- ID is candidate key of instructor
- course\_id is candidate key of course

Although several candidate keys may exist, **one of the candidate keys is selected to be the primary key.**



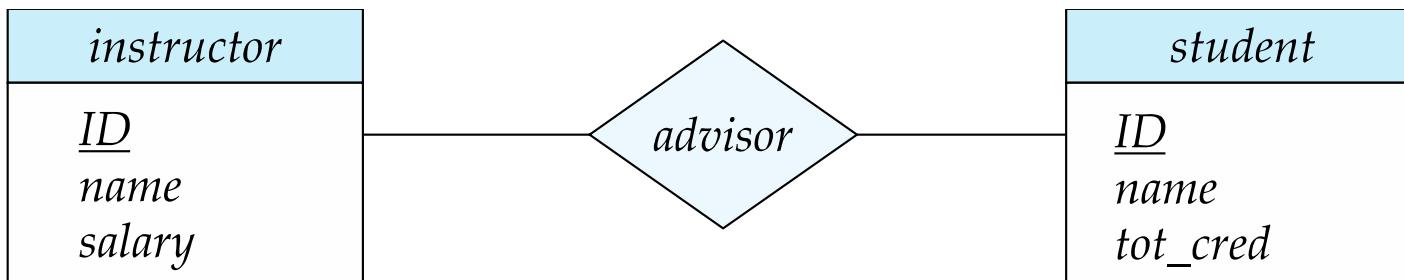
# Primary key for Entity Sets

- By definition, individual entities are distinct.
- From database perspective, the differences among them must be expressed in terms of their attributes.
- The values of the attribute values of an entity must be such that they can uniquely identify the entity.
  - No two entities in an entity set are allowed to have exactly the same value for all attributes.
- A key for an entity is a set of attributes that suffice to distinguish entities from each other



# Primary Key for Relationship Sets

- To distinguish among the various relationships of a relationship set we use the individual primary keys of the entities in the relationship set.
  - Let  $R$  be a relationship set involving entity sets  $E_1, E_2, \dots, E_n$
  - The primary key for  $R$  consists of the union of the primary keys of entity sets  $E_1, E_2, \dots, E_n$
  - If the relationship set  $R$  has attributes  $a_1, a_2, \dots, a_m$  associated with it, then the primary key of  $R$  also includes the attributes  $a_1, a_2, \dots, a_m$
- Example: relationship set “advisor”. The choice of the primary key for a relationship set depends on the mapping cardinality of the relationship set.
  - The primary key consists of  $instructor.ID$  and  $student.ID$





# Choice of Primary key for Binary Relationship

- Many-to-Many relationships. The preceding union of the primary keys is a minimal superkey and is chosen as the primary key.
- One-to-Many relationships . The primary key of the “Many” side is a minimal superkey and is used as the primary key.
- Many-to-one relationships. The primary key of the “Many” side is a minimal superkey and is used as the primary key.
- One-to-one relationships. The primary key of either one of the participating entity sets forms a minimal superkey, and either one can be chosen as the primary key.

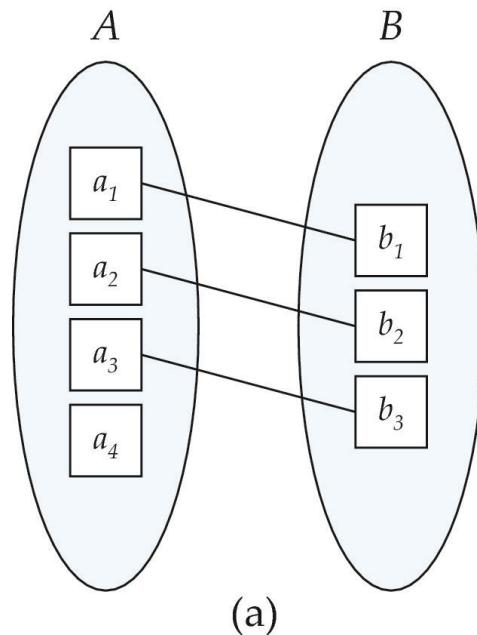


# Mapping Cardinality Constraints

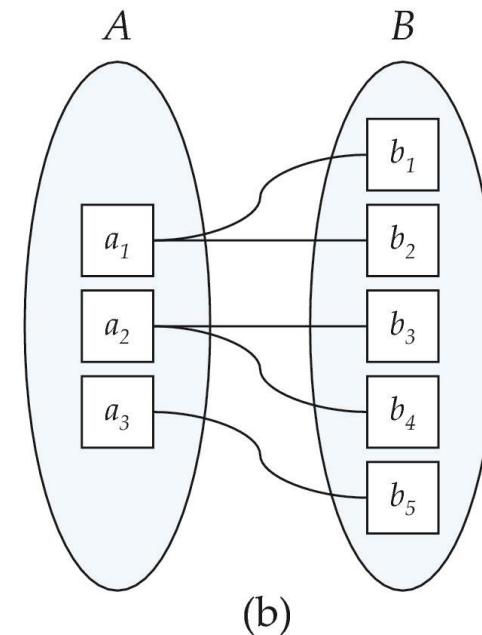
- Express the number of entities to which another entity can be associated via a relationship set.
- Most useful in describing binary relationship sets.
- For a binary relationship set the mapping cardinality must be one of the following types:
  - One to one
  - One to many
  - Many to one
  - Many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



# Mapping Cardinalities



One to one

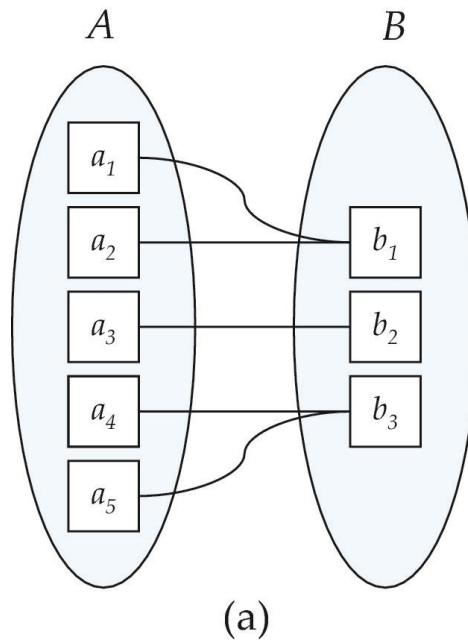


One to many

Note: Some elements in A and B may not be mapped to any elements in the other set

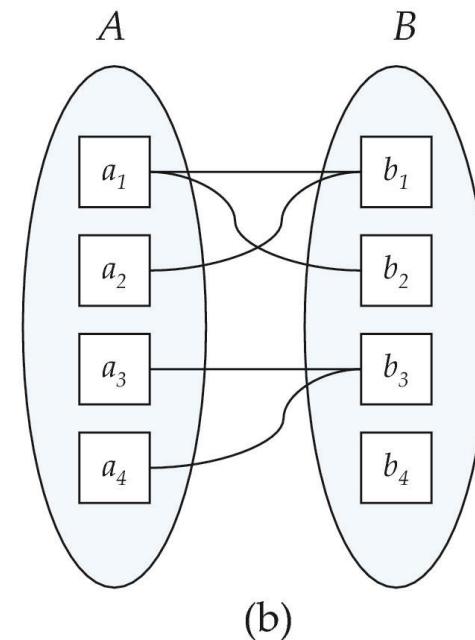


# Mapping Cardinalities



(a)

Many to one



(b)

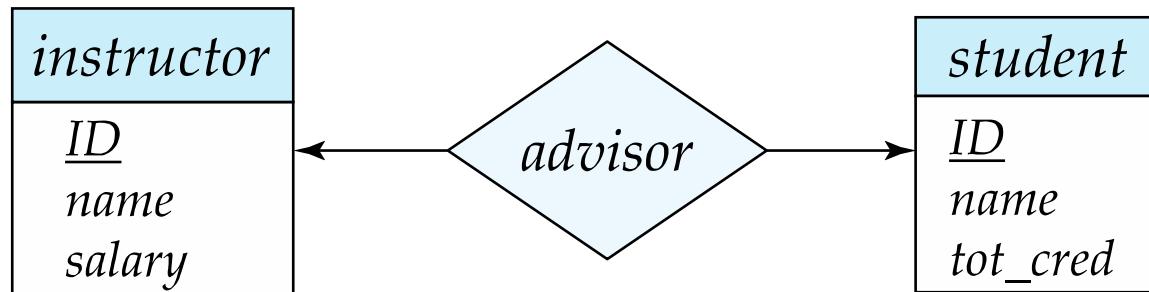
Many to many

Note: Some elements in A and B may not be mapped to any elements in the other set



# Representing Cardinality Constraints in ER Diagram

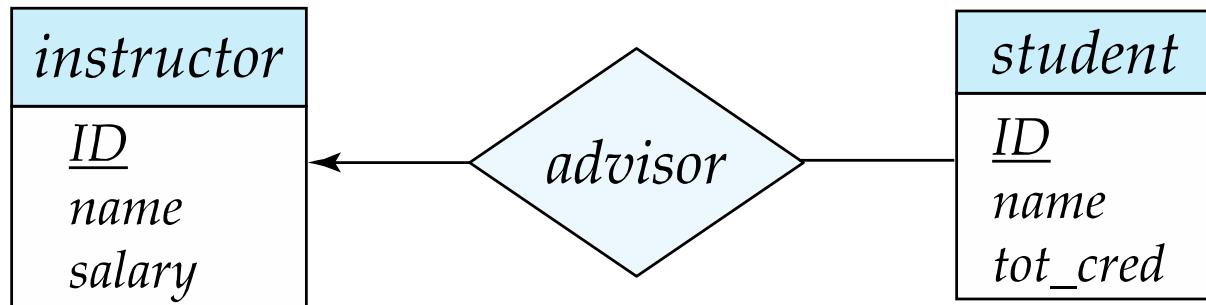
- We express cardinality constraints by drawing either a directed line ( $\rightarrow$ ), signifying “one,” or an undirected line ( $-$ ), signifying “many,” between the relationship set and the entity set.
- One-to-one relationship between an *instructor* and a *student* :
  - A student is associated with at most one *instructor* via the relationship *advisor*
  - A *student* is associated with at most one *department* via *stud\_dept*





# One-to-Many Relationship

- one-to-many relationship between an *instructor* and a *student*
  - an instructor is associated with several (including 0) students via *advisor*
  - a student is associated with at most one instructor via advisor,





# Many-to-One Relationships

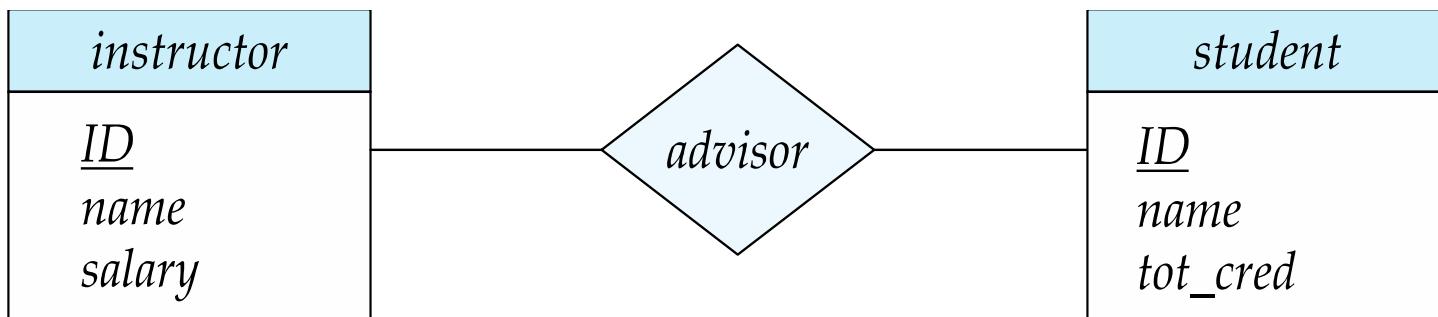
- In a many-to-one relationship between an *instructor* and a *student*,
  - an *instructor* is associated with at most one *student* via *advisor*,
  - and a *student* is associated with several (including 0) *instructors* via *advisor*





# Many-to-Many Relationship

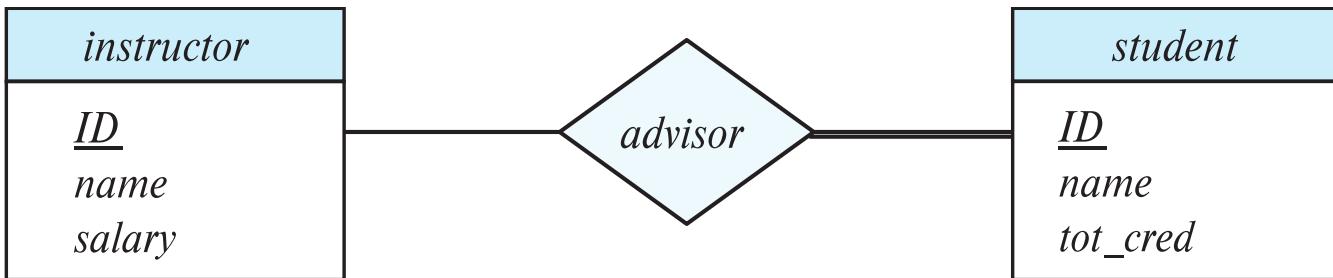
- An instructor is associated with several (possibly 0) students via *advisor*
- A student is associated with several (possibly 0) instructors via *advisor*





# Total and Partial Participation

- **Total participation** (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set



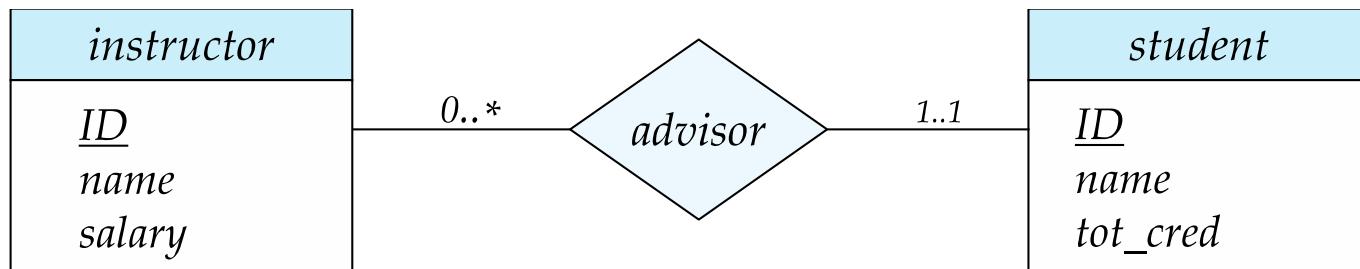
participation of *student* in *advisor* relation is total

- every *student* must have an associated *instructor*
- **Partial participation:** some entities may not participate in any relationship in the relationship set
  - Example: participation of *instructor* in *advisor* is partial



# Notation for Expressing More Complex Constraints

- A line may have an associated minimum and maximum cardinality, shown in the form  $l..h$ , where  $l$  is the minimum and  $h$  the maximum cardinality
  - A minimum value of 1 indicates total participation.
  - A maximum value of 1 indicates that the entity participates in at most one relationship
  - A maximum value of \* indicates no limit.
- Example



- Instructor can advise 0 or more students. A student must have 1 advisor; cannot have multiple advisors



# Weak Entity Sets

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



# Weak Entity Sets (Cont.)

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



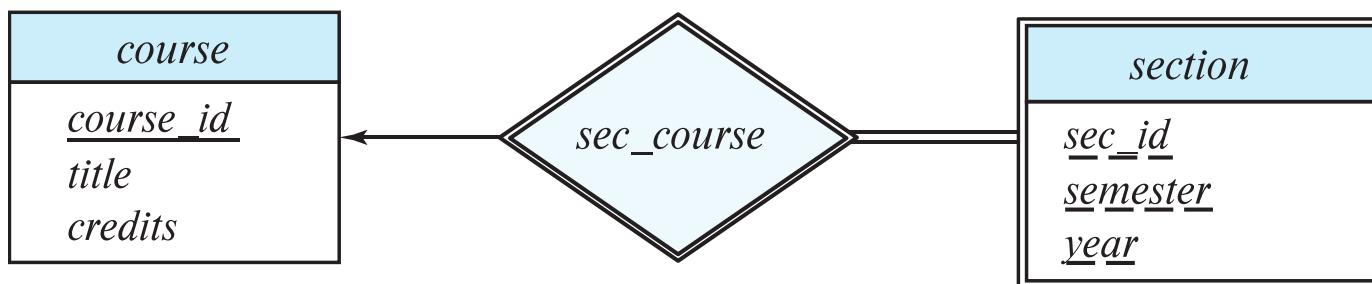
# Weak Entity Sets (Cont.)

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



# Expressing Weak Entity Sets

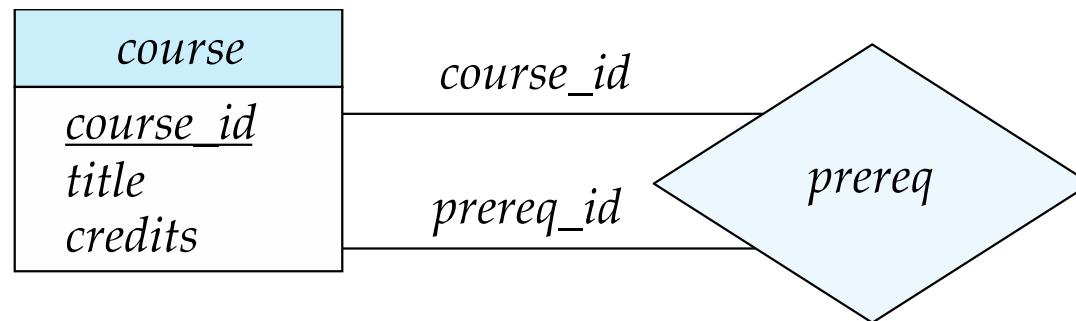
- In E-R diagrams, a weak entity set is depicted via a double rectangle.
- We underline the discriminator of a weak entity set with a dashed line.
- The relationship set connecting the weak entity set to the identifying strong entity set is depicted by a double diamond.
- Primary key for *section* –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





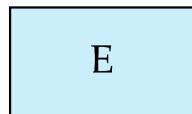
# Roles

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

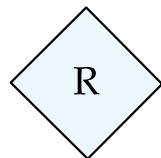




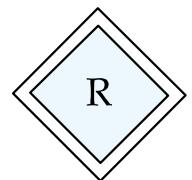
# Summary of Symbols Used in E-R Notation



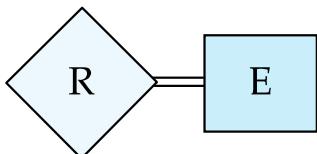
entity set



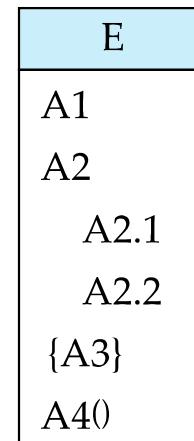
relationship set



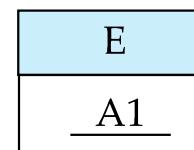
identifying  
relationship set  
for weak entity set



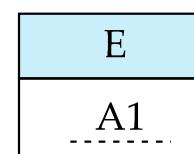
total participation  
of entity set in  
relationship



attributes:  
simple (A1),  
composite (A2) and  
multivalued (A3)  
derived (A4)



primary key



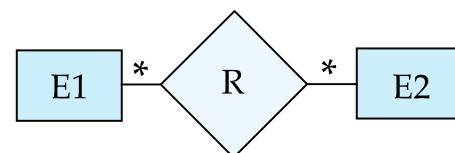
discriminating  
attribute of  
weak entity set



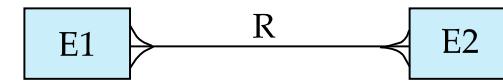
# Alternative ER Notations

Chen

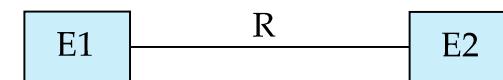
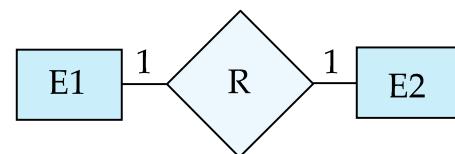
many-to-many  
relationship



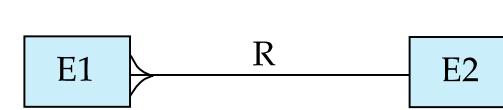
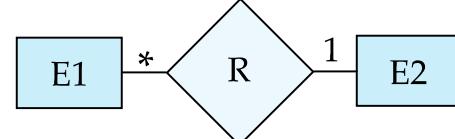
IDE1FX (Crows feet notation)



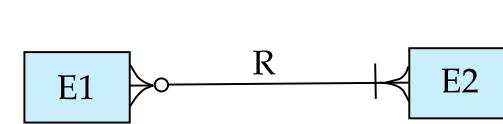
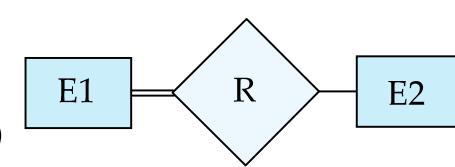
one-to-one  
relationship



many-to-one  
relationship



participation  
in R: total (E1)  
and partial (E2)



# **Logical Database Design - Mapping ERD to Relational**

# Objective

*To be able to:*

- Transform an E-R diagram to a logically equivalent set of relations

# Topics

- Transforming ER Diagrams into Relations

# Transforming ERD into Relations

Transforming (mapping) E-R diagrams to relations is a relatively straightforward process with a well-defined set of rule

Steps:

- 1: Map Regular Entities
- 2: Map Weak Entities
- 3: Map Binary Relationships
- 4: Map Unary Relationships
- 5: Map EER

# 1. Map Regular Entities

Each regular entity type in an ERD is transformed into a relation.

## Entity

entity name

simple attribute

entity identifier

composite attribute

multi-valued attribute

## Relation

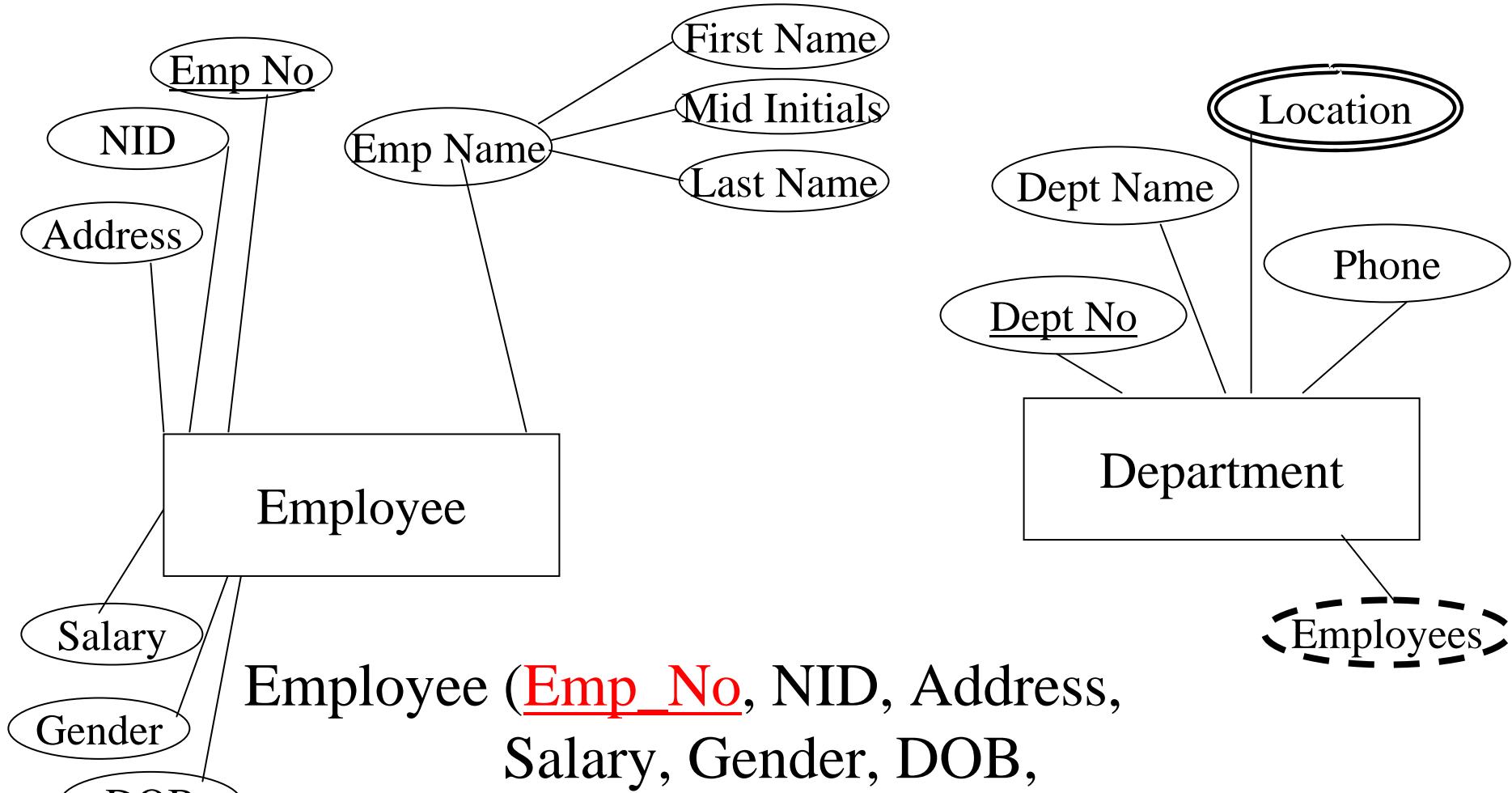
relation name

attribute of the relation

primary key of relation

component attributes

new relation with PK



Employee (Emp No, NID, Address,  
Salary, Gender, DOB,  
First\_Name, Mid\_Initials, Last\_Name)

Department(Dept No, Dept\_Name, Phone)

Dept\_Location(Dept No, Location)

FK

## 2. Map Weak Entities

Each weak entity type in an ERD is transformed into a relation.

### **Entity**

entity name

simple attribute

owner entity identifier

entity identifier (partial)

composite attribute

multi-valued attribute

### **Relation**

relation name

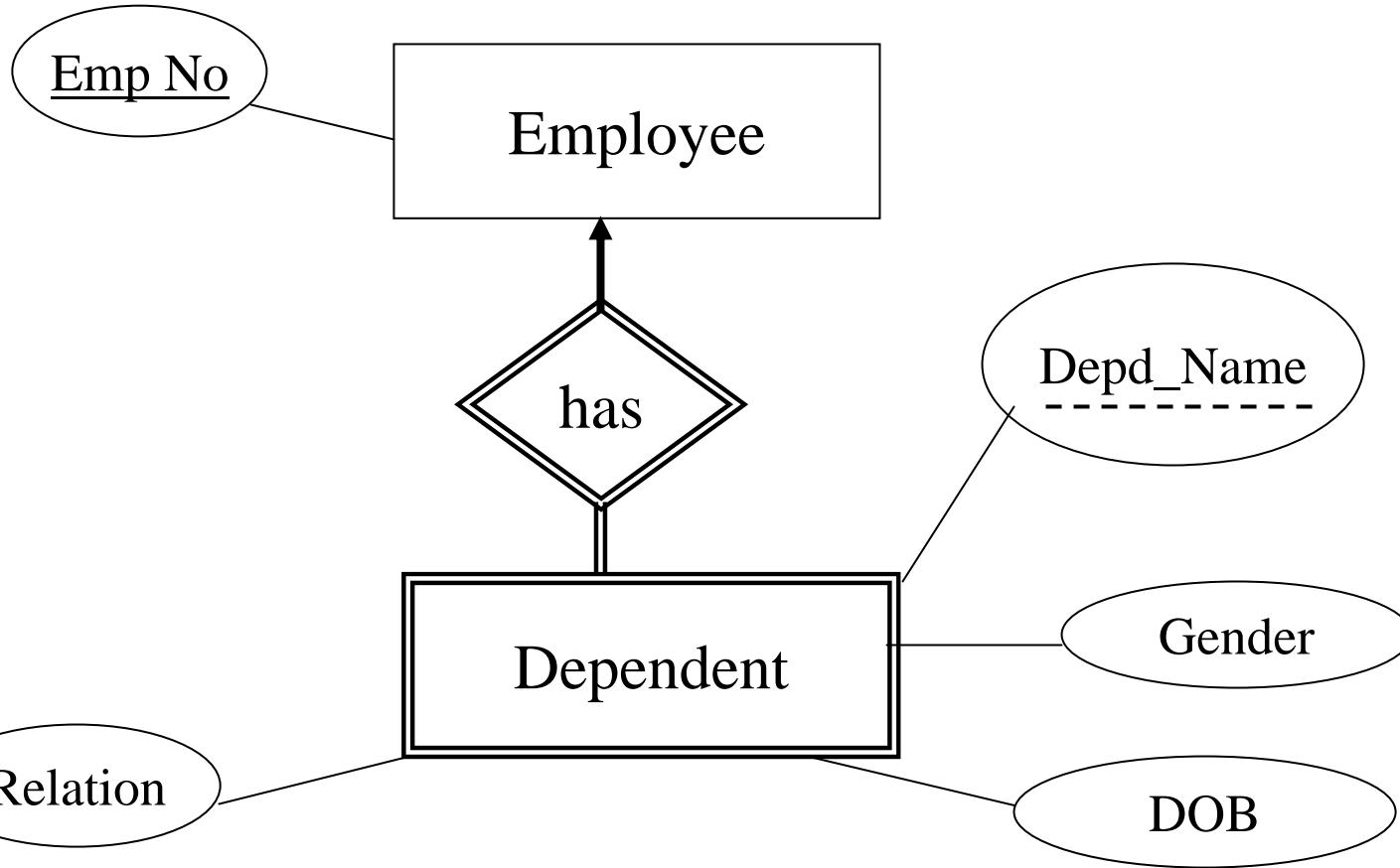
attribute of the relation

foreign key attribute

composite key together  
with PK of owner as FK

component attributes

new relation with PK



Dependent(*Emp\_No*, *Depd\_Name*, Gender, DOB, Relation)

FK

Employee (*Emp\_No*, ....)

### 3. Map Binary Relationships

Procedure depends on both the degree of the binary relationships and the cardinalities of the relationships

- Map Binary One-to-Many Relationships
- Map Binary Many-to-Many Relationships
- Map Binary One-to-One Relationships

# Map Binary One-to-Many Relationships

Create a relation for the two entity types participating in the relationships (step 1)

include PK of the entity in the one-side of the relationship as a foreign key in the relation of the many side of the relationship

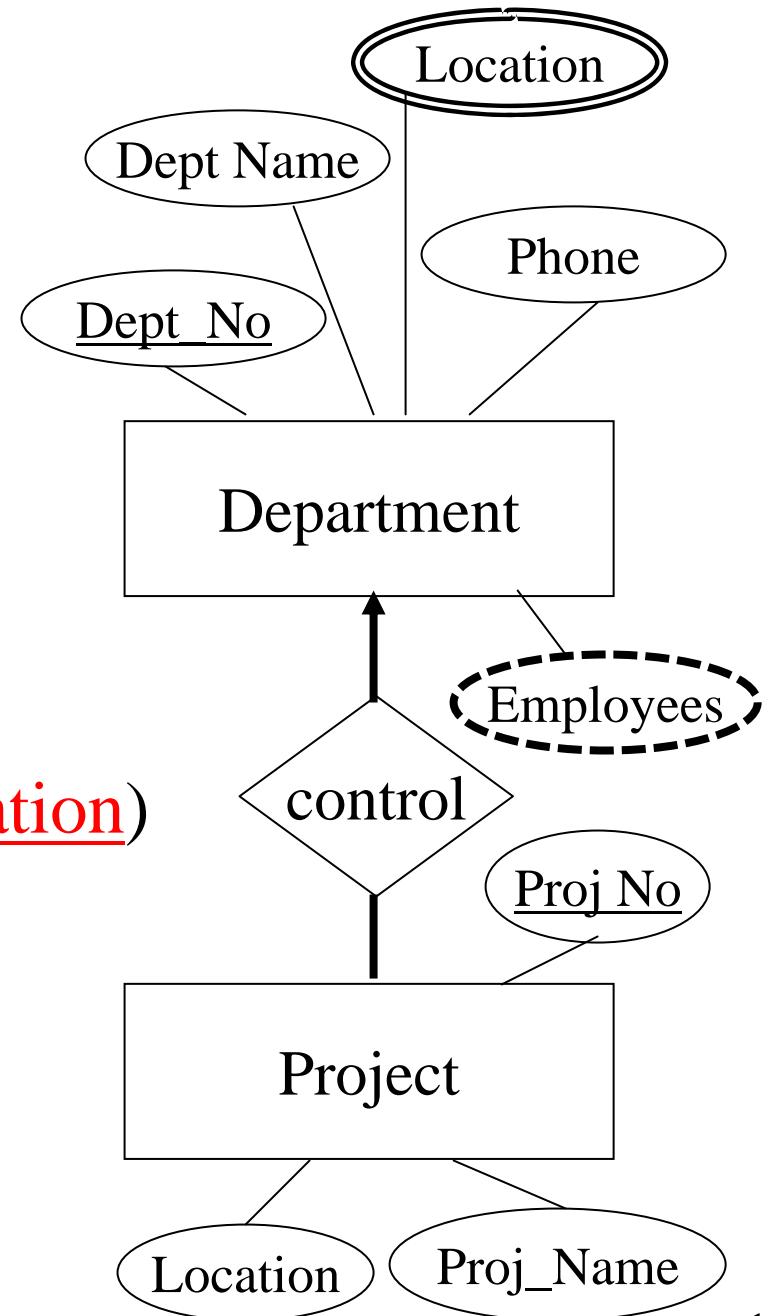
include any attributes of the relationship to the relation of the many side

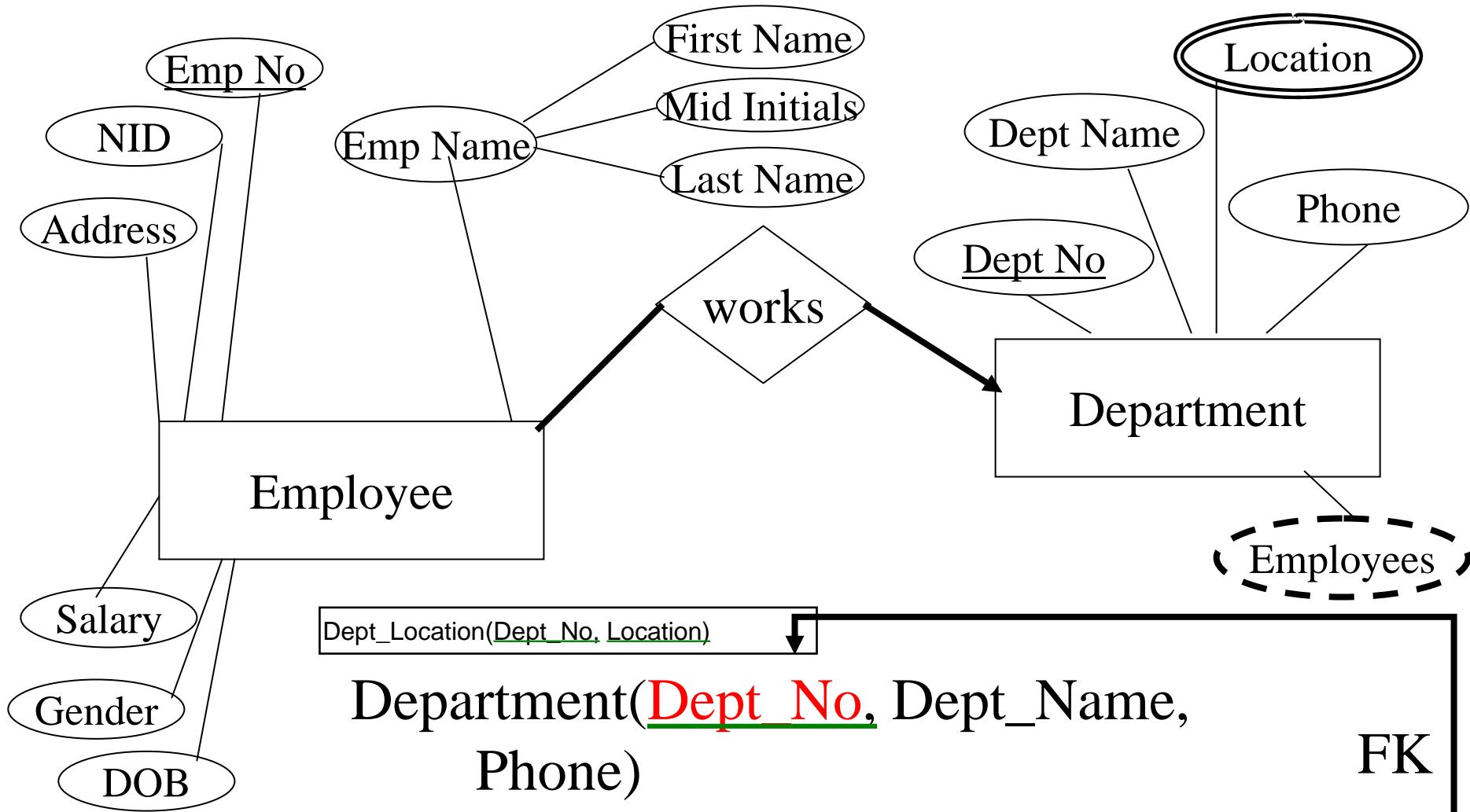
FK

Department(Dept\_No,  
Dept\_Name, Phone)

Dept\_Location(Dept\_No, Location)

Project(Proj\_No, Proj\_Name,  
Location, *Dept\_No*)





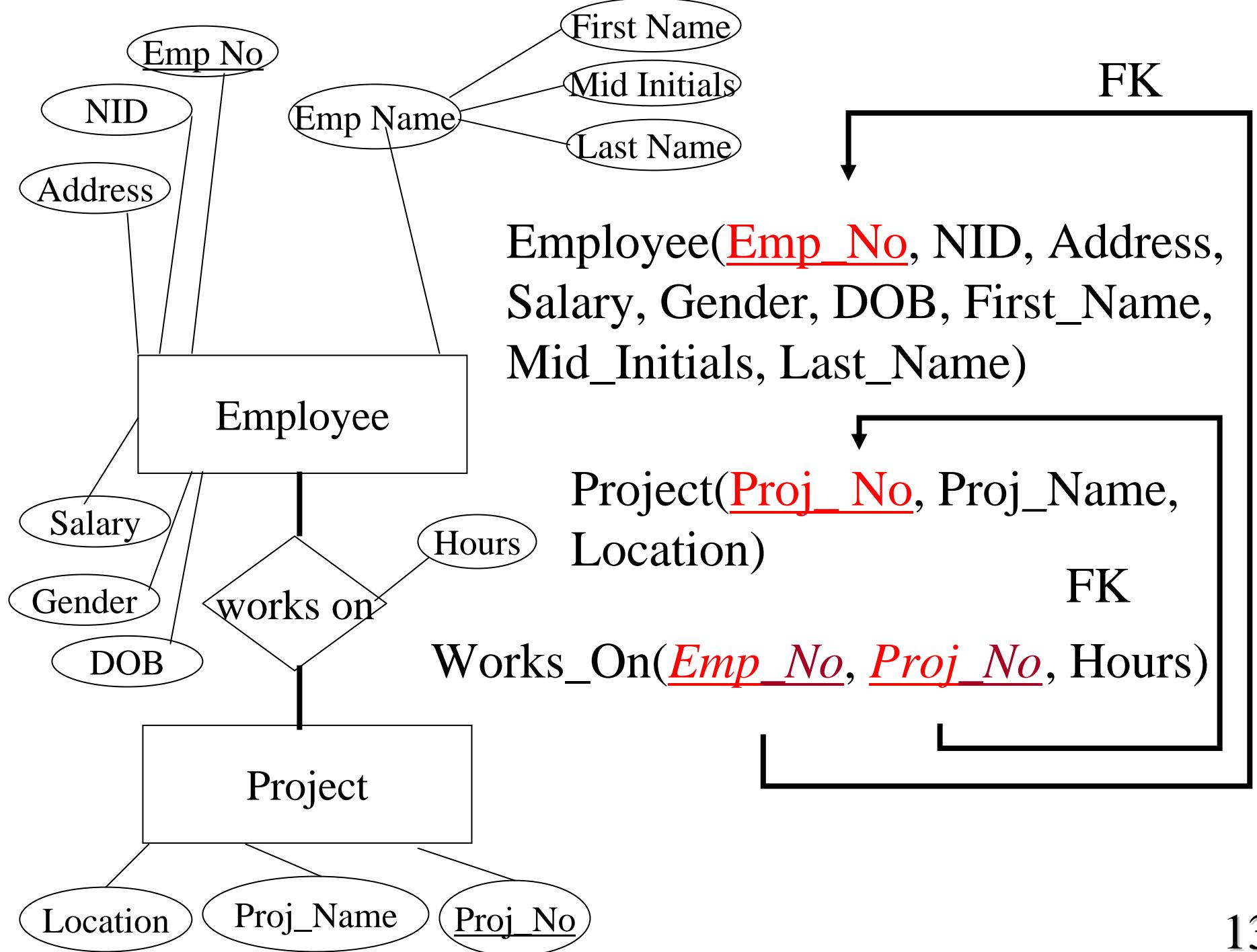
Employee(Emp\_No, NID, Address, Salary, Gender, DOB, First\_Name, Mid\_Initials, Last\_Name, *Dept\_No*)

# Map Binary Many-to-Many Relationships

Create a relation for the two entity types participating in the relationships (step 1)

Create new relation and include PK of each of the two participating entity types as FK. These attributes become the PK (composite)

include any attributes of the relationship to the new relation



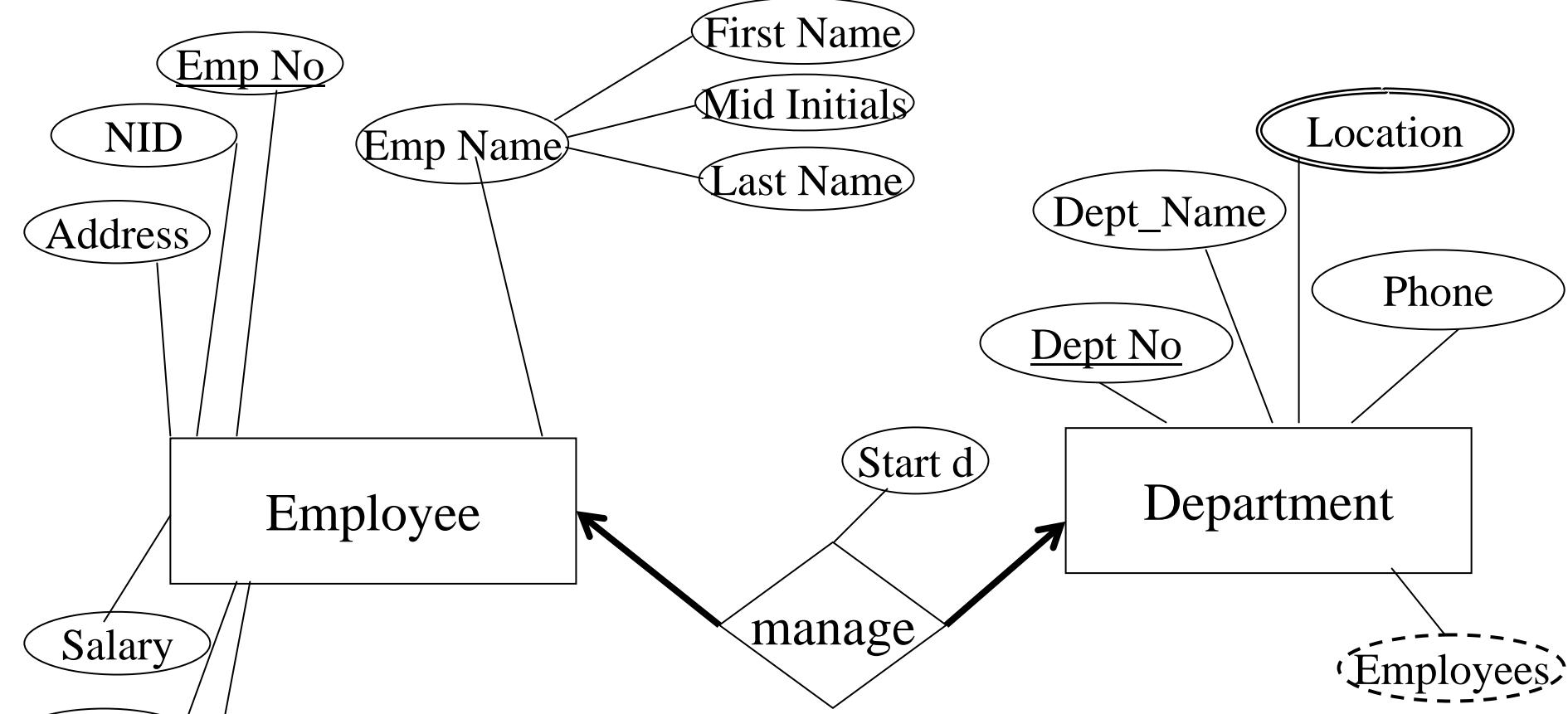
# Map Binary One-to-One Relationships

Specialised case of One-to-Many

Create a relation for the two entity types  
participating in the relationships (step 1)

Include PK of one of the relations as a  
foreign key of the other relation (include in  
optional side of the relationship that has  
mandatory participation in the 1:1)

include any attributes of the relationship to  
the same relation



Employee(Emp No, NID, Address,  
Salary, Gender, DOB, First\_Name,  
Mid\_Initials, Last\_Name)

Department(Dept No, Dept\_Name, Phone, *Manager*,  
Start\_D)

## 4. Map Unary Relationships

Procedure depends on both the degree of the binary relationships and the cardinalities of the relationships

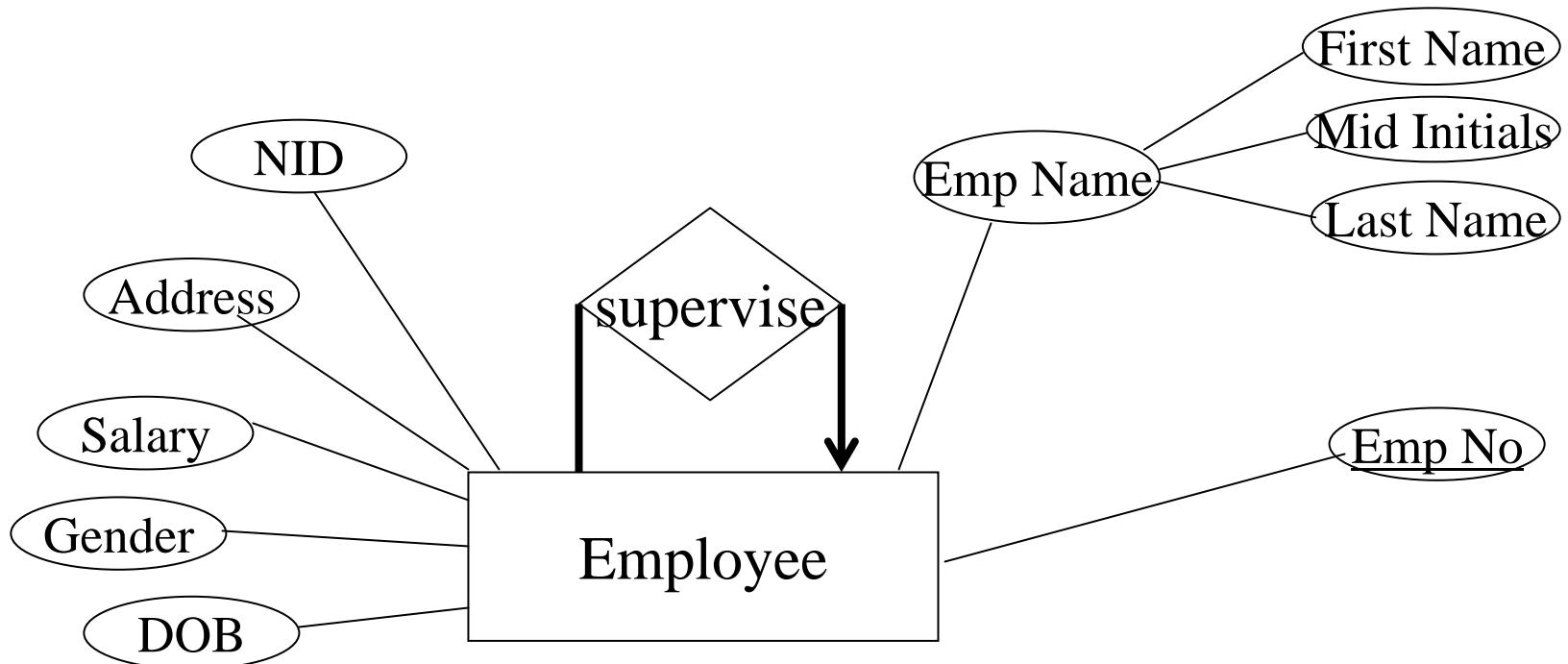
- Map Unary One-to-Many Relationships
- Map Unary Many-to-Many Relationships
- Map Unary One-to-One Relationships

# Map Unary One-to-Many Relationships

Create a relation for the entity type (step 1)

include PK of the entity as a foreign key  
within the same relation

include any attributes of the relationship to  
the relation of the many side



Employee(Emp\_No, NID, Address,  
 Salary, Gender, DOB, First\_Name,  
 Mid\_Initials, Last\_Namet\_No,  
*Supervisor*)

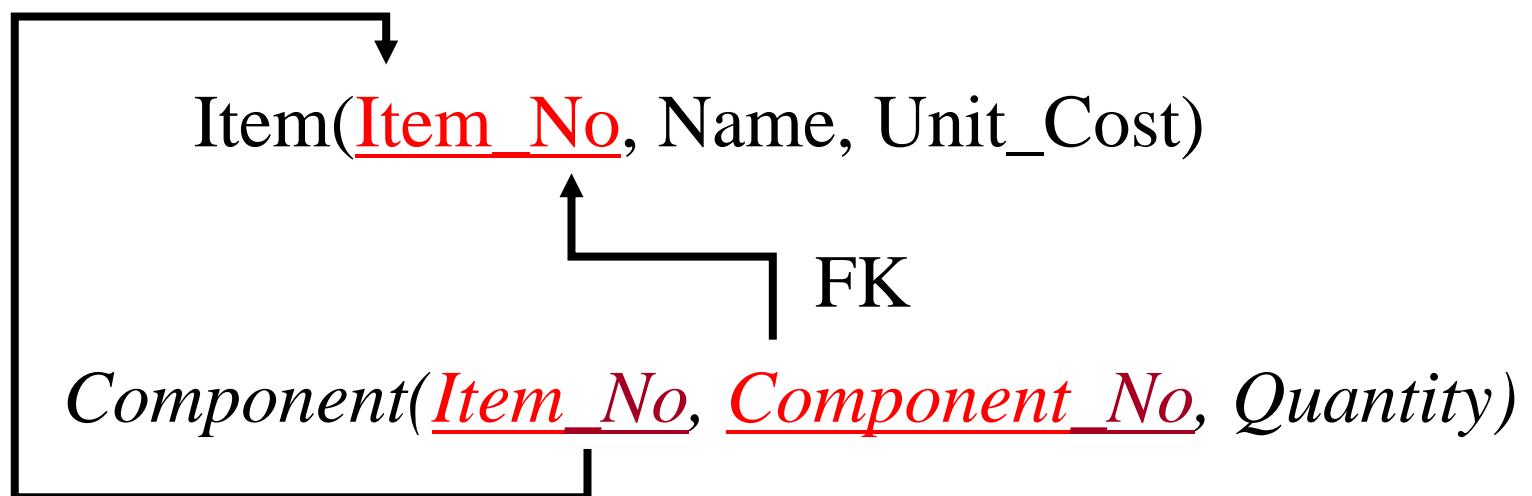
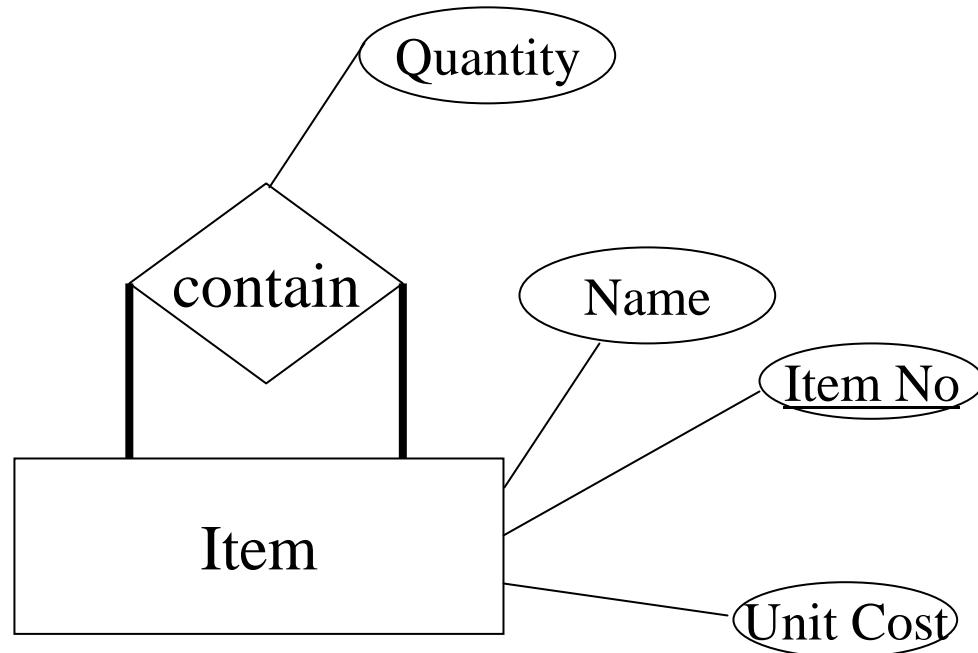
FK

# Map Unary Many-to-Many Relationships

Create a relation for the entity type (step 1)

Create new relation and include PK of the entity type as FK twice. These attributes become the PK (composite)

include any attributes of the relationship to the new relation



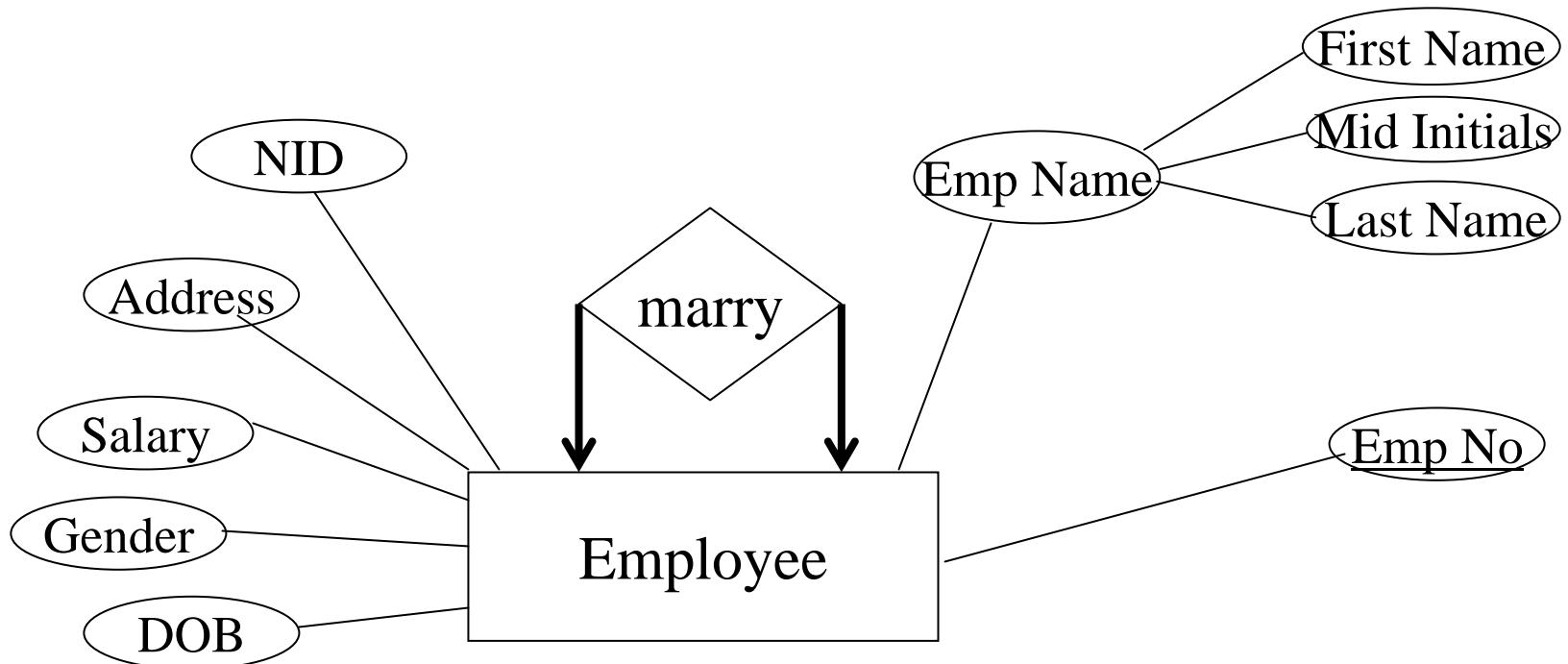
FK

# Map Unary One-to-One Relationships

Create a relation for the entity type (step 1)

include PK of the entity as a foreign key  
within the same relation

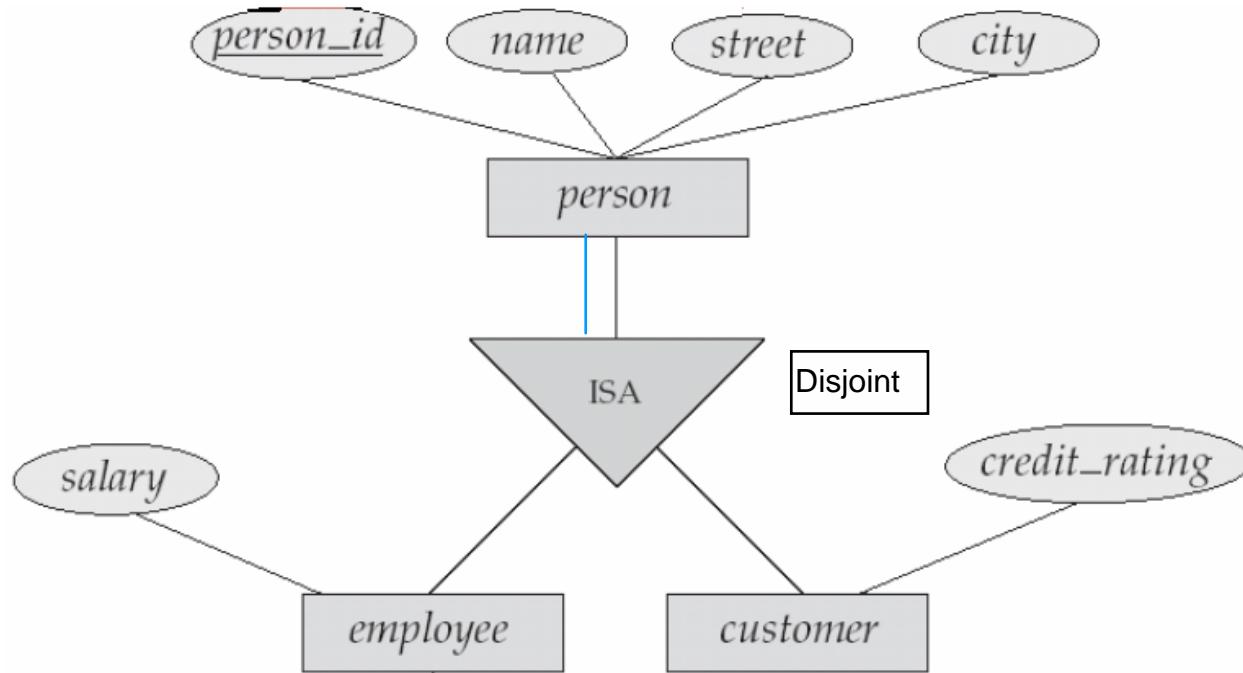
include any attributes of the relationship to  
the relation of the many side



Employee(Emp\_No, NID, Address,  
Salary, Gender, DOB, First\_Name,  
Mid\_Initials, Last\_Name,  
*Marry*)

FK

## 5. EER Mapping: Case#1 (Total and Disjoint)



We need to create 2 tables:

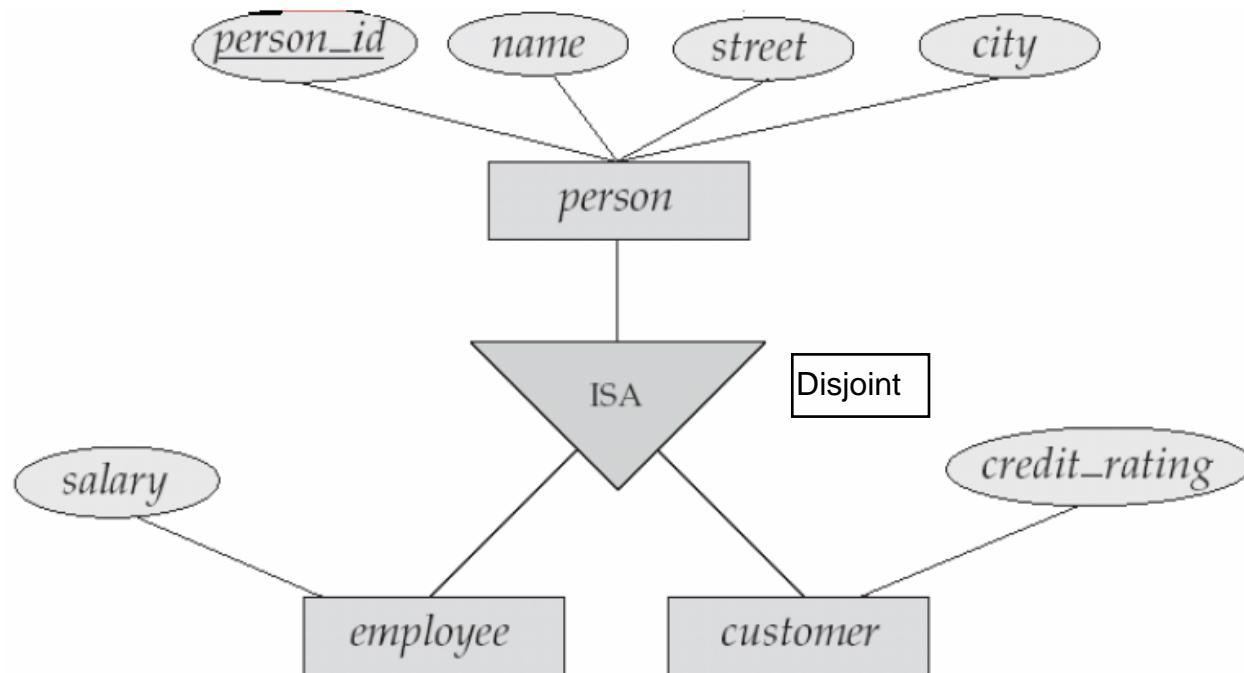
*employee* = (Person\_id, name, street, city, salary)

*Customer* = (Person\_id, name, street, city, credit\_rating)

As it is total, no need to create person table

As it is disjoint, there is no redundancy as no employee is customer and no customer is employee.

## 5. EER Mapping: Case#2 (Partial and Disjoint)



We need to create 3 tables:

*Person* = (*Person\_id*, *name*, *street*, *city*)

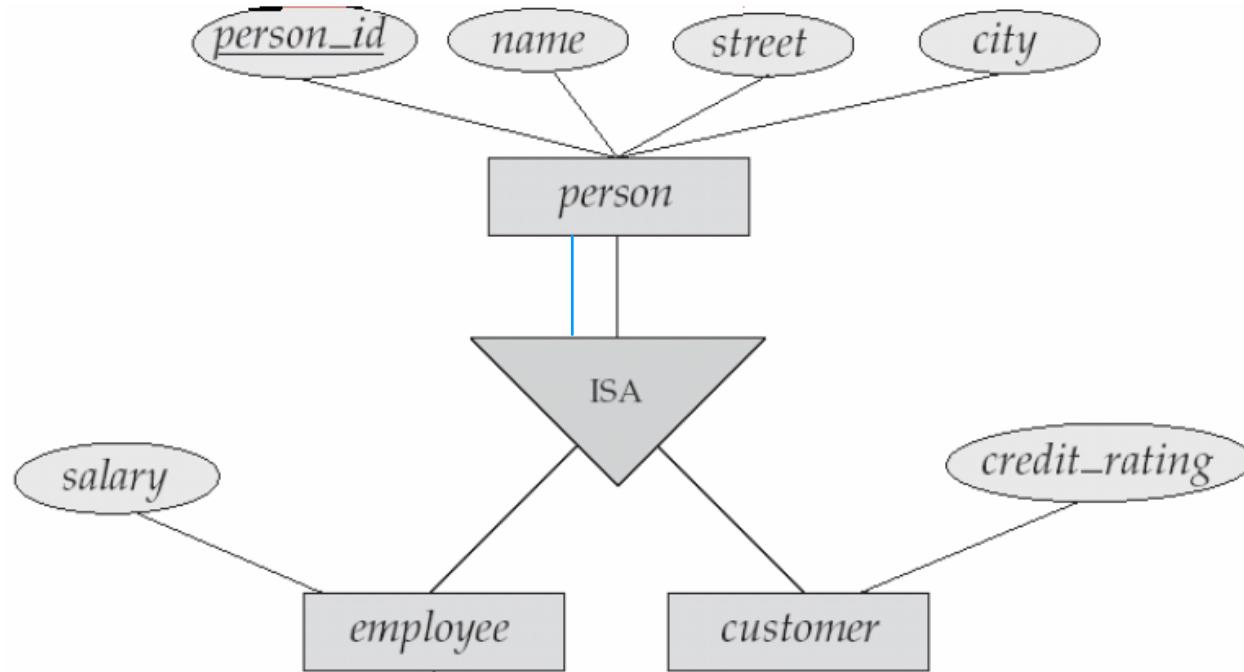
*employee* = (*Person\_id*, *name*, *street*, *city*, *salary*)

*Customer* = (*Person\_id*, *name*, *street*, *city*, *credit\_rating*)

As it is partial, we have to use person table.

As it is disjoint, there is no redundancy as no employee is customer and no customer is employee.

## 5. EER Mapping: Case#3 (Total and Overlapping)



We need to create 3 tables:

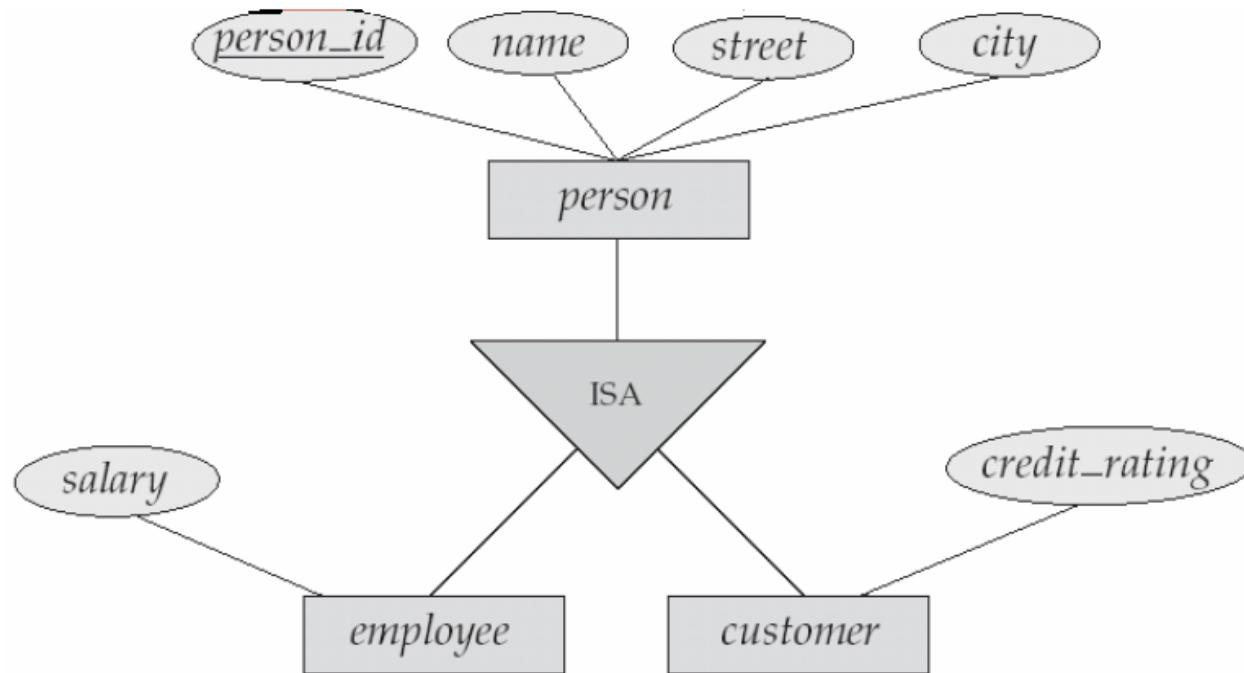
*Person* = (*Person\_id*, *name*, *street*, *city*)

*employee* = (*Person\_id*, *salary*)

*Customer* = (*Person\_id*, *credit\_rating*)

Although it is total, but we still need to create person table because otherwise, there will be some redundancy.

## 5. EER Mapping: Case#4 (Partial and Overlapping)



We need to create 3 tables:

*Person* = (*Person\_id*, *name*, *street*, *city*)

*employee* = (*Person\_id*, *salary*)

*Customer* = (*Person\_id*, *credit\_rating*)

# Normalization

## DBMS 1902224

# Database Normalization

- **Definition**
  - Optimizing table structures!
  - Removing duplicate data entries.
  - Accomplished by thoroughly investigating the various data types and their relationships with one another.
  - Follows a series of normalization “forms” or states.

# Why Normalize?

- Improved speed.
- More efficient use of space.
- Increased data integrity.

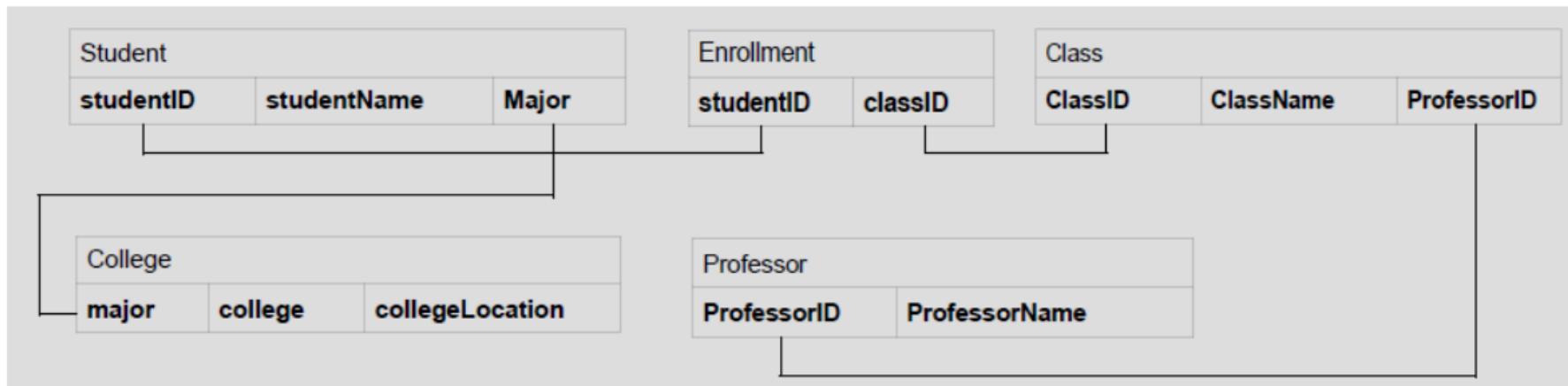
# Example!

- Old Design

How ?

Student_id	StudentName	Major	college	collegeLocation	classID	className	ProfessorID	Professor Name
999-40-9876	Ahmad	Math	Science	UJ	Math101	Discrete Math	111-2231	XYZ
999-40-9876	Ahmad	Math	Science	UJ	Skills101	Computer Skills	111-2124	YZR
999-40-9000	Salem	CIS	KASIT	UJ	CIS322	DB	111-2222	URT

- New design (After Normalization)



# Anomalies (Review)

- Anomalies are inconsistencies in data that occur due to unnecessary redundancy.
- Update anomaly.
  - Some copies of a data item are updated, but others are not.  
*e.g. update the course duration for some students*
- Insertion anomaly
  - Does not allow insertion of data unless it is accompanied by other unrelated data. e.g., *Table student and courses. Can't insert new student unless he/she is related with some course*
- Deletion anomaly
  - Can't delete some data without also deleting other, unrelated data. *e.g. deleting a student will also delete data of course*

# Data Normalization

- Primarily a tool to validate and improve a logical design so that it satisfies certain constraints that avoid unnecessary duplication of data.
- The process of decomposing relations with anomalies to produce smaller, well-structured relations.

# Anomaly Examples

OID	ODate	CID	CName
1	8/10/2004	2	ABC Inc
2	8/10/2004	2	ABC Inc
3	8/11/2004	3	XZY Co
4	8/11/2004	4	QWE Inc
5	8/12/2004	4	QWE Inc

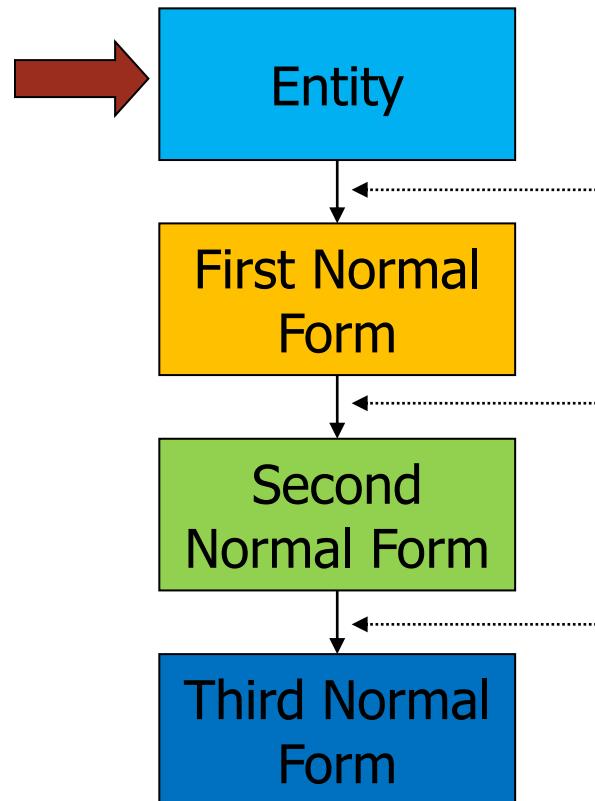
Add a customer: Must also add an order

Update the name of Customer #2: Must update multiple rows

Delete Order #3: Must also delete info about Customer #3

# Steps in Normalization

(doesn't meet the definition of a relation)



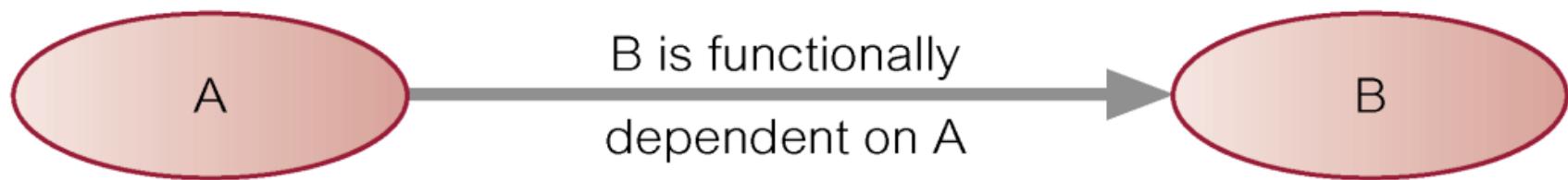
Remove multivalued & composite attributes.  
Meet definition of relation

Remove partial dependencies

Remove transitive dependencies

# *Functional Dependency*

Definition: “*relationship between or among attributes*”



- Attribute B is functionally dependent on A.

# Functional Dependency: An Example!

$\text{StudentID} \rightarrow \text{StudentName}$

$\text{classID} \rightarrow \text{className}$

$(\text{StudentID}, \text{classID}) \rightarrow \text{Grade}$

StudentID	StudentName	ClassID	ClassName	Grade
St_1	Ahmad Salem	1902322	Database Design	A
St_1	Ahmad Salem	1902321	Data Security	B+
St_2	Osama Othman	1902322	Database Design	C+
St_3	Ameer Asem	1902322	Database Design	A

# First Normal Form (1NF)

If a table of data meets the definition of a relation, it is in first normal form.

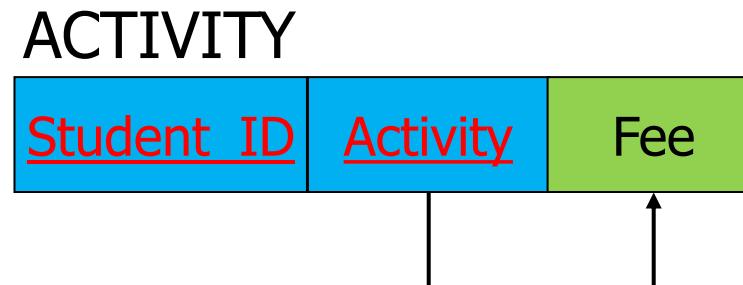
- Every relation has a unique name.
- Every attribute value is atomic (single-valued).
- Every row is unique.
- Attributes in tables have unique names.
- The order of the columns is irrelevant.
- The order of the rows is irrelevant.

# Second Normal Form (2NF)

- 1NF and no partial functional dependencies.
- Partial functional dependency: when one or more non-key attributes are functionally dependent on part of the primary key.
- Every non-key attribute must be defined by the entire key, not just by part of the key.
- If a relation has a single attribute as its key, then it is automatically in 2NF.

# Second Normal Form (2NF)

A relation  
that is not  
in 2NF



Key: Student\_ID, Activity  
 $\text{Activity} \rightarrow \text{Fee}$

Fee is determined by Activity

<b>Student_ID</b>	<b>Activity</b>	<b>Fee</b>
222-22-2020	Swimming	30
232-22-2111	Golf	100
222-22-2020	Golf	100
255-24-2332	Hiking	50

# Second Normal Form (2NF)

**Divide the relation into two relations that now meet 2NF**

**STUDENT\_ACTIVITY**

<u>Student_ID</u>	Activity
-------------------	----------

Key: Student\_ID and Activity

**ACTIVITY\_COST**

<u>Activity</u>	Fee
-----------------	-----

Key: Activity

Activity → Fee

<b>Student_ID</b>	<b>Activity</b>
222-22-2020	Swimming
232-22-2111	Golf
222-22-2020	Golf
255-24-2332	Hiking

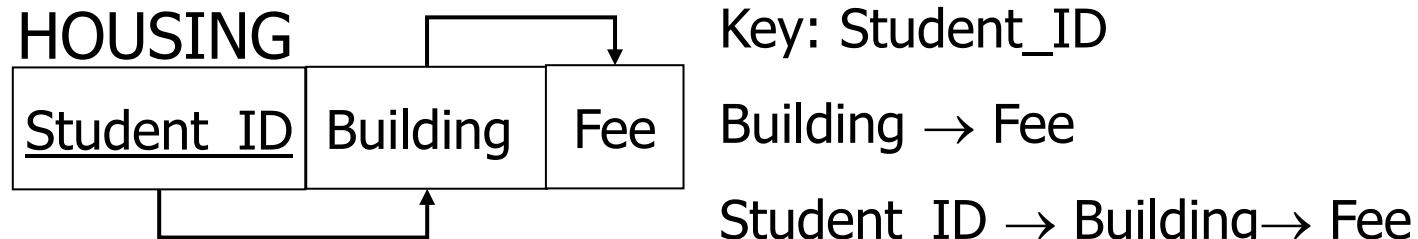
<b>Activity</b>	<b>Fee</b>
Swimming	30
Golf	100
Hiking	50

# Third Normal Form (3NF)

- 2NF and no transitive dependencies
- Transitive dependency: a functional dependency between two or more non-key attributes.

# Third Normal Form (3NF)

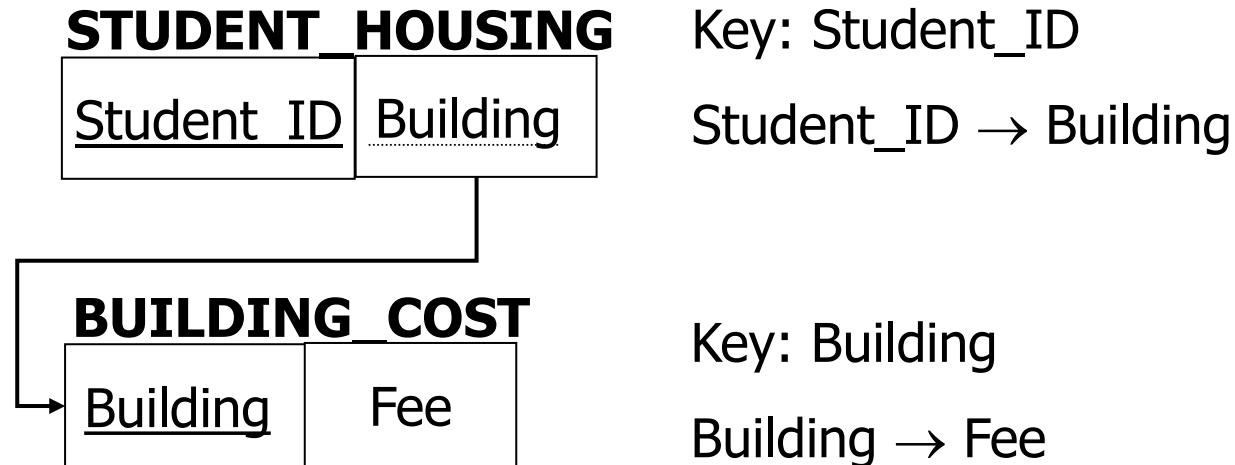
A relation  
with a  
transitive  
dependency



<b>Student_ID</b>	<b>Building</b>	<b>Fee</b>
222-22-2020	Dabney	1200
232-22-2111	Liles	1000
222-22-5554	The Range	1100
255-24-2332	Dabney	1200
330-25-7789	The Range	1100

# Third Normal Form (3NF)

Divide the relation into two relations that now meet 3NF



Key: Student\_ID

Student\_ID → Building

Key: Building

Building → Fee

<b>Student_ID</b>	<b>Building</b>
222-22-2020	Dabney
232-22-2111	Liles
222-22-5554	The Range
255-24-2332	Dabney
330-25-7789	The Range

<b>Building</b>	<b>Fee</b>
Dabney	1200
Liles	1000
The Range	1100

# Case Study 2

PORJ_NUM	Proj_Name	EMP_Num	Emp_Name	Job_class	Charge_hours	Hours_worked
123	UJ-Proj	17	XYZ	Engineer	JD 100	35
		90	UHY	Designer	JD 150	21
		50	OKI	Admin	JD 400	39
389	KASIT-Proj	90	UHY	Designer	JD 150	17
		65	PLU	Worker	JD 75	40
		83	RGT	Worker	JD 75	40
		24	KUH	Engineer	JD 100	30

# Case Study 2: to 1<sup>st</sup> NF

PORJ_NUM	Proj_Name	EMP_Num	Emp_Name	Job_class	Charge_hours	Hours_worked
123	UJ-Proj	17	XYZ	Engineer	JD 100	35
		90	UHY	Designer	JD 150	21
		50	OKI	Admin	JD 400	39
389	KASIT-Proj	90	UHY	Designer	JD 150	17
		65	PLU	Worker	JD 75	40
		83	RGT	Worker	JD 75	40
		24	KUH	Engineer	JD 100	30



Elimination of repeated Groups

Table 1: PORJ\_NUM

Proj\_Name

Table 2:

PORJ\_NUM

EMP\_Num

Emp\_Name

Job\_class

Charge\_hours

Hours\_worked

# Case Study 2: to 2<sup>nd</sup> NF

Table 1:	<u>PORJ_NUM</u>	Proj_Name				
Table 2:	<u>PORJ_NUM</u>	<u>EMP_Num</u>	Emp_Name	Job_class	Charge_hours	Hours_worked

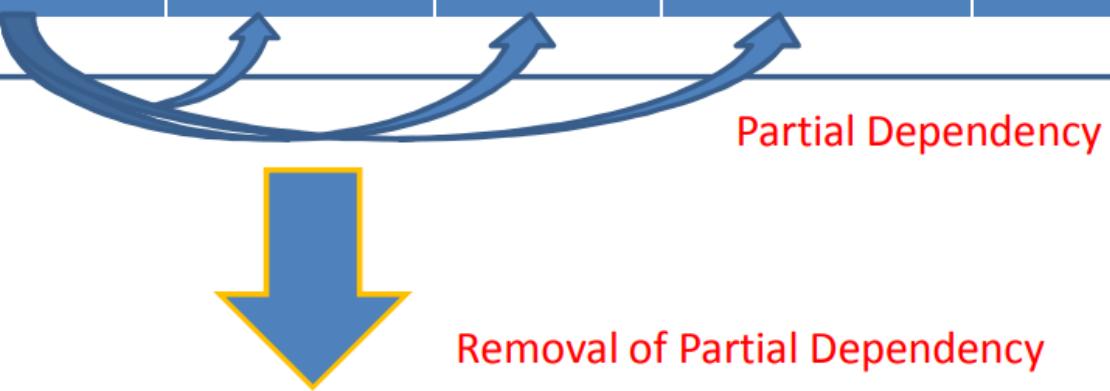


Table 1:	<u>PORJ_NUM</u>	Proj_Name		
Table 2:	<u>EMP_Num</u>	Emp_Name	Job_class	Charge_hours
Table 3:	<u>PORJ_NUM</u>	<u>EMP_Num</u>	Hours_worked	

# Case Study 2: to 3<sup>rd</sup> NF

Table 1: PORJ\_NUM | Proj\_Name

Table 2: EMP\_Num | Emp\_Name | Job\_class | Charge\_hours

Table 3: PORJ\_NUM | EMP\_Num | Hours\_worked



Transitive Dependency



Removal of Transitive Dependency

Table 1: PORJ\_NUM | Proj\_Name

Table 2: EMP\_Num | Emp\_Name | Job\_class

Table 3: PORJ\_NUM | EMP\_Num | Hours\_worked

Table 4: Job\_class | Charge\_hours

# Case Study 2: DB in 3<sup>rd</sup> NF

ProjInfo

<u>PORJ_NUM</u>	Proj_Name
-----------------	-----------

ProjEmpl

<u>PORJ_NUM</u>	<u>EMP_Num</u>	Hours_worked
-----------------	----------------	--------------



Employee

<u>EMP_Num</u>	Emp_Name	Job_class
----------------	----------	-----------

Jobs

<u>Job_class</u>	Charge_hours
------------------	--------------



# Case Study 3 (slide 4)

Student_id	StudentName	Major	college	collegeLocation	classID	className	ProfessorID	Professor Name
999-40-9876	Ahmad	Math	Science	UJ	Math101	Discrete Math	111-2231	XYZ
999-40-9876	Ahmad	Math	Science	UJ	Skills101	Computer Skills	111-2124	YZR
999-40-9000	Salem	CIS	KASIT	UJ	CIS322	DB	111-2222	URT

# Database Development Lifecycle (DDL and DML)

---

## ■ Summary of Development Steps

1. Create a Data Model from gathered requirements.
  - Entity-Relationship Model
2. Transpose Data Model into Relation(s)
3. Normalize Relations
4. Create the Relational Schema within Database Management System.  
(Metadata)
  - **Create the database (we are here)**
  - Create the table(s)

# Commands to Create Database

---

- To shows the databases on the system:  
use command: SHOW DATABASES;

- **To create a database:**

- General Syntax Format:

```
CREATE DATABASE databaseName;
```

```
CREATE SCHEMA schemaName;
```

- Examples: CREATE DATABASE test;  
CREATE SCHEMA test;

# Commands to Create Database

## ■ To remove a database:

- General Syntax Format:

```
DROP DATABASE databaseName;
```

```
DROP SCHEMA databaseName;
```

- Examples:

```
DROP DATABASE test;
```

```
DROP SCHEMA test;
```

- Can add IF EXISTS to prevent the DROP DATABASE from executing if the database does not exist.

- Examples: `DROP DATABASE IF EXISTS test;`

- `DROP SCHEMA IF EXISTS test;`

# USE command

---

- Once a MySQL database is created, it isn't automatically opened for access within the session
- “Open” a database through USE command
  - General Syntax Format: `USE databaseName;`
  - Example: `USE test;`
- Only one MySQL database can be opened at a time in a session
- To open a different database in the session, execute a USE command

# Database Development Lifecycle

---

## ■ Summary of Development Steps

1. Create a Data Model from gathered requirements.
  - Entity-Relationship Model
2. Transpose Data Model into Relation(s)
3. Normalize Relations
4. Create the Relational Schema within Database Management System.  
(Metadata)
  - Create the database
  - **Create the tables (we are here now)**

# Attribute Data Types

---

## ■ Character

- CHAR (*size*) –fixed-length string
- VARCHAR (*size*) –variable-length string

## ■ Numeric

### ■ Integer

- INT or INTEGER

## ■ Date/Time

- DATE
- YYYY-MM-DD

# What is a constraint?

---

- A constraint is a rule limiting the values allowed for an attribute
- A PRIMARY KEY is one example
  - Each value must be unique
  - A value must be specified; NULL values are not allowed
- There are other constraints that protect data integrity
- Constraints are specified in the CREATE TABLE statement

# DDL: CREATE | DROP TABLE

## ■ To create a table within the current database (based on USE command)

### ■ Syntax Format (so far):

```
CREATE TABLE tableName (  
    attribute1 DATATYPE,  
    attribute2 DATATYPE,  
    CONSTRAINTS: PRIMARY KEY() and FOREIGN KEY()  
);
```

### ■ Example:

```
CREATE TABLE employee (  
    employeeID CHAR(11),  
    Dept CHAR(11),  
    name VARCHAR(35),  
    PRIMARY KEY(employeeID,  
    FOREIGN KEY (DEPT) REFERENCES DEPARTMENT(DEPt_NAME));
```

## ■ To remove a table within the current database

### ■ `DROP TABLE tableName;`

# ALTER TABLE

- Now that you can create a table, if you need to change the structure, you can alter it instead of dropping and creating again.
- Databases are no different than programs
  - Version 1 is just the starting point
  - There will be many versions as time goes on
- There are lots of changes that can be made through ALTER TABLE. The syntax is shown in MySQL Help System
  - <https://dev.mysql.com/doc/refman/8.0/en/alter-table.html>

# What to Alter?

---

- You can use the ALTER TABLE statement to:
  - Add/Drop a primary key constraint
  - Add/Modify/Drop a column
  - Add/Drop/Modify a default value
  
- You cannot modify a table level constraint...

# Altering Primary Key

---

- Let's get rid of the PRIMARY KEY constraint:

```
ALTER TABLE film
```

```
DROP PRIMARY KEY;
```

- Add it back

```
ALTER TABLE film
```

```
ADD PRIMARY KEY film_pk (filmID);
```

# ADD | DROP Columns

- General Syntax Format:

```
ALTER TABLE tableName
{ADD | DROP} COLUMN columnspec;
```

- Example: What if we wanted to add a “URL” column to the film table:

```
ALTER TABLE film
ADD COLUMN url VARCHAR(50);
```

- Example: What if we wanted to remove the replacement cost from the film table:

```
ALTER TABLE film
DROP COLUMN replacementCost;
```

# Altering Defaults

- Defaults are the only “property” you can alter.
  - We will see in the next slide how to change data types.

- Let’s add a default value of ‘Not entered.’ to the description column in the table:

```
ALTER TABLE film
```

```
ALTER description SET DEFAULT ‘Not entered.’ ;
```

- And if we change our mind and don’t want it after all:

```
ALTER TABLE film
```

```
ALTER description DROP DEFAULT;
```

# ALTER TABLE...MODIFY COLUMN

- Through the ALTER TABLE...MODIFY statement, you can edit an existing attribute's specification including changing the data type or adding/removing a not null constraint or default value
  - ALTER TABLE *tableName*  
→ MODIFY *attribute DATATYPE [NOT NULL] [DEFAULT];*
  - CHANGE (instead of modify) can also be used:  
→ CHANGE currAttrName newAttrName newDATATYPE;
  - Note that we need to state the FULL attribute specification, not just what we want to change
- Example:  
ALTER TABLE film  
MODIFY COLUMN releaseYear SMALLINT NOT NULL DEFAULT 2020;

# Commands to show metadata

---

- `SHOW DATABASES;`
  - Shows the databases on the system
- `SHOW TABLES;`
  - Shows the tables in the current database
- `DESCRIBE tableName;`
  - Shows some of the metadata maintained for the specified table
- `DESC tableName;`
- `EXPLAIN tableName;`
- `SHOW COLUMNS FROM tableName;`
- `SHOW CREATE TABLE tableName;`

# DML: INSERT

---

- Adds user data to a table
- The `INSERT` statement typically adds one record at a time
  - Most DBMS packages also have a bulk insert capability (e.g., Oracle has SQLLoader)
- General Syntax Format:

```
INSERT INTO tableName  
(fieldlist) VALUES (valuelist);
```

- *tableName* is the name of the table we're adding to
- *fieldlist* is a comma-separated list of attribute names
  - Provides a template so that the DBMS engine know which attributes you are specifying values for and what attribute order the values will be given in
- *valuelist* is a comma-separated list of values to be added for the attributes listed in the *fieldlist*

# INSERT Shortcut

```
INSERT INTO employee  
(employeeID, ssn, name, dob)  
VALUES ('111111111111',  
'121-21-2121', 'Jason Jones',  
'1990-10-31');
```

```
INSERT INTO employee  
VALUES ( '222222222222',  
'342-56-2454',  
'Lisa Kellog',  
'1984-01-15');
```

You may specify only the values if you know the correct order of the attributes in the table AND will be specifying a value for each attribute.

# INSERT Shortcut

```
INSERT INTO employee  
VALUES ('12345678901', '254-67-4563');
```

**WRONG!**

```
INSERT INTO employee (employeeID,  
ssn) VALUES ('12345678901',  
'254-67-4563');
```

**CORRECT**

When using the shortcut and attributes are NOT specified,  
ALL values must be specified in the CORRECT ORDER.

# Inserting NULL values

- To see if NULLs are allowed for an attribute, DESCRIBE the table it is in
- Can be used with or without field list
  - Examples:

```
INSERT INTO employee (employeeID, ssn, name)  
VALUES ('33333333333', '098-76-5432', NULL);
```

```
INSERT INTO employee VALUES ('44444444444',  
'555-44-3333', NULL, null);
```

# Common Mistakes with NULL

---

- `INSERT INTO employee VALUES ('777777777777', '656-56-5656', '', NULL);`
- `INSERT INTO employee VALUES ('8888888888', '444-99-1111', ' ', NULL);`
- `INSERT INTO employee VALUES ('9999999999', '777-55-0909', 'NULL', NULL);`

# DML: UPDATE

- Modifies existing user data

- General Syntax Format:

    UPDATE *tableName*

        SET    *field1 = value1, ..., fieldN = valueN*

        WHERE *condition*;    <- WHERE clause is optional

- *tableName* is the name of the table we're updating
- *condition* is an expression to limit which records are updated
- *field1...fieldN* are the names of the fields to be updated
- *value1...valueN* are the new values

- Example:

    UPDATE film

        SET replacementCost = replacementCost +5

        WHERE replacementCost=9.99;

# DML: DELETE

- Removes record(s) from a table

- General Syntax Format:

DELETE

FROM *tableName*

WHERE *condition(s)*; <- WHERE clause is optional

- *tableName* is the name of the table we're deleting records from
- *condition* is an expression to limit which records are deleted (similar to WHERE in the SELECT statement)
  - Without WHERE all data is deleted from the table

- Example:

DELETE

FROM film

WHERE rating='R';

# MySQL Scripts

- MySQL allows you to specify more than one SQL statement at a time
  - Always use a semi-colon at the end of the statement
- MySQL also allows you to run SQL statements that are contained in a text file (also known as a script file)
- Executing a script file has advantages over typing statements at the command line
  - Typos are easily corrected
  - Scripts can be saved and reused
  - Commenting can be included
    - -- single line comment (space MUST be included after --)
    - # single line comment
    - /\* block comment (can span multiple lines) \*/
    - COMMENT “Comment within an attribute specification”
      - See COMMENT for employeeID on previous slide

# MySQL Scripts

```
mysql>.\ c:\FILE_PATH\file.sql
```

command      space      Full File Path      No semi-colon

Or

```
mysql> SOURCE c:\path\file.sql
```



# Chapter 3: Introduction to SQL

Database System Concepts, 7<sup>th</sup> Ed.

©Silberschatz, Korth and Sudarshan

See [www.db-book.com](http://www.db-book.com) for conditions on re-use



# Outline

- Overview of The SQL Query Language
- SQL Data Definition
- Basic Query Structure of SQL Queries
- Additional Basic Operations
- Set Operations
- Null Values
- Aggregate Functions
- Nested Subqueries
- Modification of the Database



# Basic Query Structure

- A typical SQL query has the form:

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

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



# The select Clause

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

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



# The select Clause (Cont.)

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

```
select distinct dept_name  
from instructor
```

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

```
select all dept_name  
from instructor
```

<i>dept_name</i>
Comp. Sci.
Finance
Music
Physics
History
Physics
Comp. Sci.
History
Finance
Biology
Comp. Sci.
Elec. Eng.



# The select Clause (Cont.)

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

```
select *
  from instructor
```

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

```
select '437'
```

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

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

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

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

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



# The select Clause (Cont.)

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

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

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

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

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



# The where Clause

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

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

- SQL allows the use of the logical connectives **and**, **or**, and **not**
- The operands of the logical connectives can be expressions involving the comparison operators <, <=, >, >=, =, and <>.
- Comparisons can be applied to results of arithmetic expressions
- To find all instructors in Comp. Sci. dept with salary > 70000

```
select name  
from instructor  
where dept_name = 'Comp. Sci.' and salary > 70000
```

<i>name</i>
Katz
Brandt



# The from Clause

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

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

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



# Examples

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

<i>name</i>	<i>course_id</i>
Srinivasan	CS-101
Srinivasan	CS-315
Srinivasan	CS-347
Wu	FIN-201
Mozart	MU-199
Einstein	PHY-101
El Said	HIS-351
Katz	CS-101
Katz	CS-319
Crick	BIO-101
Crick	BIO-301
Brandt	CS-190
Brandt	CS-190
Brandt	CS-319
Kim	EE-181



# The Rename Operation

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

*old-name as new-name*

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

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

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

*instructor as T ≡ instructor T*



# String Operations

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

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

- Match the string “100%”

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

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



# String Operations (Cont.)

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



# Ordering the Display of Tuples

- List in alphabetic order the names of all instructors

```
select distinct name  
from instructor  
order by name
```

- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
  - Example: **order by name desc**
- Can sort on multiple attributes
  - Example: **order by dept\_name, name**



# Where Clause Predicates

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



# Set Operations

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

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

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

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

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

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



# Set Operations (Cont.)

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



# Null Values

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



# Null Values (Cont.)

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



# Aggregate Functions

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

**avg:** average value

**min:** minimum value

**max:** maximum value

**sum:** sum of values

**count:** number of values



# Aggregate Functions Examples

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



# Aggregate Functions – Group By

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

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

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



# Aggregation (Cont.)

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

- /\* erroneous query \*/

```
select dept_name, ID, avg (salary)
from instructor
group by dept_name;
```



# Aggregate Functions – Having Clause

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

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

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



# Nested Subqueries

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

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

as follows:

- **From clause:**  $r_i$  can be replaced by any valid subquery
- **Where clause:**  $P$  can be replaced with an expression of the form:  
$$B <\text{operation}> (\text{subquery})$$

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

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



# Set Membership



# Set Membership

- Find courses offered in Fall 2017 and in Spring 2018

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

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

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



# Set Membership (Cont.)

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

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

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

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

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



# Set Comparison



# Set Comparison – “some” Clause

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

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

- Same query using > **some** clause

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



# Definition of “some” Clause

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

$(5 < \text{some} \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{true}$  (read: 5 < some tuple in the relation)

$(5 < \text{some} \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{false}$

$(5 = \text{some} \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{true}$

$(5 \neq \text{some} \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{true}$  (since  $0 \neq 5$ )

$(= \text{some}) \equiv \text{in}$

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



# Set Comparison – “all” Clause

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

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



# Definition of “all” Clause

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

$(5 < \text{all} \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{false}$

$(5 < \text{all} \begin{array}{|c|} \hline 6 \\ \hline 10 \\ \hline \end{array}) = \text{true}$

$(5 = \text{all} \begin{array}{|c|} \hline 4 \\ \hline 5 \\ \hline \end{array}) = \text{false}$

$(5 \neq \text{all} \begin{array}{|c|} \hline 4 \\ \hline 6 \\ \hline \end{array}) = \text{true} \text{ (since } 5 \neq 4 \text{ and } 5 \neq 6\text{)}$

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

However,  $(= \text{all}) \neq \text{in}$



# Test for Empty Relations

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



# Use of “exists” Clause

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

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

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



# Use of “not exists” Clause

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

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

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



# Test for Absence of Duplicate Tuples

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

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



# Subqueries in the From Clause



# Subqueries in the Form Clause

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

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

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

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



# 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



# End of Chapter 3



# Chapter 4 : Intermediate SQL

Database System Concepts, 7<sup>th</sup> Ed.

©Silberschatz, Korth and Sudarshan

See [www.db-book.com](http://www.db-book.com) for conditions on re-use



# Outline

- Join Expressions
- Views
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- Index Definition in SQL
- Authorization



# Joined Relations

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



# Natural Join in SQL

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



# Natural Join in SQL (Cont.)

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

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



# Student Relation

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



# Takes Relation

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



# *student natural join takes*

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



# Dangerous in Natural Join

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

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

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



# Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples from one relation that does not match tuples in the other relation to the result of the join.
- Uses *null* values.
- Three forms of outer join:
  - left outer join
  - right outer join
  - full outer join



# Outer Join Examples

- Relation *course*

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

- Relation *prereq*

<i>course_id</i>	<i>prereq_id</i>
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

- Observe that

*course* information is missing CS-347

*prereq* information is missing CS-315



# Left Outer Join

- course **natural left outer join** prereq

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>	<i>prereq_id</i>
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	<i>null</i>

- In relational algebra:  $\text{course} \bowtie \text{prereq}$



# Right Outer Join

- course **natural right outer join** prereq

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	null	null	null	CS-101

- In relational algebra: course  $\bowtie$  prereq



# Full Outer Join

- course **natural full outer join** prereq

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	null
CS-347	null	null	null	CS-101

- In relational algebra: course  $\bowtie$  prereq



# Joined Types and Conditions

- **Join operations** take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the **from** clause
- **Join condition** – defines which tuples in the two relations match.
- **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

<i>Join types</i>
<b>inner join</b>
<b>left outer join</b>
<b>right outer join</b>
<b>full outer join</b>

<i>Join conditions</i>
<b>natural</b>
<b>on &lt; predicate &gt;</b>
<b>using (A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>n</sub>)</b>



# Joined Relations – Examples

- course **natural right outer join prereq**

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	null	null	null	CS-101

- course **full outer join prereq using (course\_id)**

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	null
CS-347	null	null	null	CS-101



# Joined Relations – Examples

- **course inner join prereq on**  
 $course.course\_id = prereq.course\_id$

course_id	title	dept_name	credits	prereq_id	course_id
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190

- What is the difference between the above, and a natural join?
- **course left outer join prereq on**  
 $course.course\_id = prereq.course\_id$

course_id	title	dept_name	credits	prereq_id	course_id
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190
CS-315	Robotics	Comp. Sci.	3	null	null



# Joined Relations – Examples

- **course natural right outer join prereq**

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	null	null	null	CS-101

- **course full outer join prereq using (course\_id)**

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	null
CS-347	null	null	null	CS-101



# Views

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

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

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



# View Definition

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

```
create view v as < query expression >
```

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

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



# View Definition and Use

- A view of instructors without their salary

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

- Find all instructors in the Biology department

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

- Create a view of department salary totals

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



# Views Defined Using Other Views

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



# Views Defined Using Other Views

- **create view *physics\_fall\_2017* as**  
**select course.course\_id, sec\_id, building, room\_number**  
**from course, section**  
**where course.course\_id = section.course\_id**  
**and course.dept\_name = 'Physics'**  
**and section.semester = 'Fall'**  
**and section.year = '2017';**
- **create view *physics\_fall\_2017\_watson* as**  
**select course\_id, room\_number**  
**from *physics\_fall\_2017***  
**where building= 'Watson';**



# Transactions

- A **transaction** consists of a sequence of query and/or update statements and is a “unit” of work
- The SQL standard specifies that a transaction begins implicitly when an SQL statement is executed.
- The transaction must end with one of the following statements:
  - **Commit work.** The updates performed by the transaction become permanent in the database.
  - **Rollback work.** All the updates performed by the SQL statements in the transaction are undone.
- Atomic transaction
  - either fully executed or rolled back as if it never occurred
- Isolation from concurrent transactions



# 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



# Constraints on a Single Relation

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



# Not Null Constraints

- **not null**

- Declare *name* and *budget* to be **not null**

*name varchar(20) not null*

*budget numeric(12,2) not null*



# Unique Constraints

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



# 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')))
```



# 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**



# 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



# 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



# Built-in Data Types in SQL

- **date:** Dates, containing a (4 digit) year, month and date
  - Example: **date** '2005-7-27'
- **time:** Time of day, in hours, minutes and seconds.
  - Example: **time** '09:00:30'      **time** '09:00:30.75'
- **timestamp:** date plus time of day
  - Example: **timestamp** '2005-7-27 09:00:30.75'
- **interval:** period of time
  - Example: **interval** '1' day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values



# 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);
```



# 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'));
```



# Index Creation

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



# Index Creation Example

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

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

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



# 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<sub>1</sub>, U<sub>2</sub>, U<sub>3</sub>**
- <privilege-list> may be **all** to revoke all privileges the revoker may hold.
- If <revoker-list> includes **public**, all users lose the privilege except those granted it explicitly.
- If the same privilege was granted twice to the same user by different 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!



# End of Chapter 4



# Chapter 4 : Intermediate SQL

Database System Concepts, 7<sup>th</sup> Ed.

©Silberschatz, Korth and Sudarshan

See [www.db-book.com](http://www.db-book.com) for conditions on re-use



# Outline

- Join Expressions
- Views
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- Index Definition in SQL
- Authorization



# Joined Relations

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



# Natural Join in SQL

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



# Natural Join in SQL (Cont.)

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

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



# Student Relation

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



# Takes Relation

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



# *student natural join takes*

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



# Dangerous in Natural Join

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

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

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



# Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples from one relation that does not match tuples in the other relation to the result of the join.
- Uses *null* values.
- Three forms of outer join:
  - left outer join
  - right outer join
  - full outer join



# Outer Join Examples

- Relation *course*

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

- Relation *prereq*

<i>course_id</i>	<i>prereq_id</i>
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

- Observe that

*course* information is missing CS-347

*prereq* information is missing CS-315



# Left Outer Join

- course **natural left outer join** prereq

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>	<i>prereq_id</i>
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	<i>null</i>

- In relational algebra:  $\text{course} \bowtie \text{prereq}$



# Right Outer Join

- course **natural right outer join** prereq

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	null	null	null	CS-101

- In relational algebra: course  $\bowtie$  prereq



# Full Outer Join

- course **natural full outer join** prereq

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	null
CS-347	null	null	null	CS-101

- In relational algebra: course  $\bowtie$  prereq



# Joined Types and Conditions

- **Join operations** take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the **from** clause
- **Join condition** – defines which tuples in the two relations match.
- **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

<i>Join types</i>
<b>inner join</b>
<b>left outer join</b>
<b>right outer join</b>
<b>full outer join</b>

<i>Join conditions</i>
<b>natural</b>
<b>on &lt; predicate &gt;</b>
<b>using (A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>n</sub>)</b>



# Joined Relations – Examples

- course **natural right outer join prereq**

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	null	null	null	CS-101

- course **full outer join prereq using (course\_id)**

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	null
CS-347	null	null	null	CS-101



# Joined Relations – Examples

- **course inner join prereq on**  
 $course.course\_id = prereq.course\_id$

course_id	title	dept_name	credits	prereq_id	course_id
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190

- What is the difference between the above, and a natural join?
- **course left outer join prereq on**  
 $course.course\_id = prereq.course\_id$

course_id	title	dept_name	credits	prereq_id	course_id
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190
CS-315	Robotics	Comp. Sci.	3	null	null



# Joined Relations – Examples

- **course natural right outer join prereq**

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	null	null	null	CS-101

- **course full outer join prereq using (course\_id)**

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	null
CS-347	null	null	null	CS-101



# Views

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

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

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



# View Definition

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

```
create view v as < query expression >
```

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

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



# View Definition and Use

- A view of instructors without their salary

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

- Find all instructors in the Biology department

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

- Create a view of department salary totals

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



# Views Defined Using Other Views

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



# Views Defined Using Other Views

- **create view *physics\_fall\_2017* as**  
**select course.course\_id, sec\_id, building, room\_number**  
**from course, section**  
**where course.course\_id = section.course\_id**  
**and course.dept\_name = 'Physics'**  
**and section.semester = 'Fall'**  
**and section.year = '2017';**
- **create view *physics\_fall\_2017\_watson* as**  
**select course\_id, room\_number**  
**from *physics\_fall\_2017***  
**where building= 'Watson';**



# Transactions

- A **transaction** consists of a sequence of query and/or update statements and is a “unit” of work
- The SQL standard specifies that a transaction begins implicitly when an SQL statement is executed.
- The transaction must end with one of the following statements:
  - **Commit work.** The updates performed by the transaction become permanent in the database.
  - **Rollback work.** All the updates performed by the SQL statements in the transaction are undone.
- Atomic transaction
  - either fully executed or rolled back as if it never occurred
- Isolation from concurrent transactions



# 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



# Constraints on a Single Relation

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



# Not Null Constraints

- **not null**

- Declare *name* and *budget* to be **not null**

*name varchar(20) not null*

*budget numeric(12,2) not null*



# Unique Constraints

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



# 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')))
```



# 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**



# 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



# 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



# Built-in Data Types in SQL

- **date:** Dates, containing a (4 digit) year, month and date
  - Example: **date** '2005-7-27'
- **time:** Time of day, in hours, minutes and seconds.
  - Example: **time** '09:00:30'      **time** '09:00:30.75'
- **timestamp:** date plus time of day
  - Example: **timestamp** '2005-7-27 09:00:30.75'
- **interval:** period of time
  - Example: **interval** '1' day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values



# 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);
```



# 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'));
```



# Index Creation

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



# Index Creation Example

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

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

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



# 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<sub>1</sub>, U<sub>2</sub>, U<sub>3</sub>**
- <privilege-list> may be **all** to revoke all privileges the revoker may hold.
- If <revoker-list> includes **public**, all users lose the privilege except those granted it explicitly.
- If the same privilege was granted twice to the same user by different 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*?



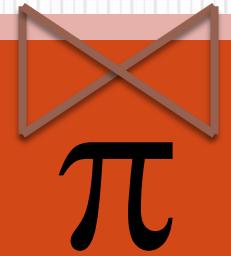
# Other Authorization Features

- **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!



# End of Chapter 4

# Relational Algebra



# Relational Query Languages

- Query languages QL: Allow manipulation and **retrieval of data** from a database.
- Relational model supports simple, powerful QLs:
  - Strong formal foundation based on logic.
  - Allows for much optimization.
- Query Languages  **$\neq$**  programming languages!
  - QLs not intended to be used for complex calculations.
  - QLs support easy, efficient access to large data sets.

# Basics of Querying

- A query is applied to relation instances, and the result of a query is also a relation instance.

R1

select ename, salary from R1 →

R2

eid	ename	Salary	age
28	Eric	90K	35
58	Kyle	100K	33



ename	Salary
Eric	90K
Kyle	100K

# Relational Algebra: 5 Basic Operations

- Selection ( $\sigma$ ) Selects a subset of *rows* from relation (horizontal).
- Projection ( $\pi$ ) Retains only wanted *columns* from relation (vertical).
- Cross-product ( $\times$ ) Allows us to combine two relations.
- Set-difference ( $-$ ) Tuples in r1, but not in r2.
- Union ( $\cup$ ) Tuples in r1 or in r2.

## Example Instances

*Boats*

<u>bid</u>	bname	color
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

*s1*

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

*s2*

<u>sid</u>	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

<i>R1</i>	<u>sid</u>	<u>bid</u>	<u>day</u>
	22	101	10/10/96
	58	103	11/12/96

# Projection ( $\pi$ )

- Examples:  $\pi_{age}(S2)$

$$\pi_{sname, rating}(S2)$$

- Retains only attributes that are in the “*projection list*”.
- Projection operator has to *eliminate duplicates*
  - Note: real systems typically don’t do duplicate elimination unless the user explicitly asks for it.

# Projection ( $\pi$ )

<u>sid</u>	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

S2

sname	rating
yuppy	9
lubber	8
guppy	5
rusty	10

$\pi_{sname, rating}(S2)$

age
35.0
55.5

$\pi_{age}(S2)$

# Selection ( $\sigma$ )

- Selects rows that satisfy *selection condition*.
  - Result is a relation.
- Schema* of result is same as that of the input relation.
- Do we need to do duplicate elimination?

$$\sigma_{rating > 8}(S2)$$

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

$$\pi_{sname, rating}(\sigma_{rating > 8}(S2))$$

sname	rating
yuppy	9
rusty	10

# Union, Intersection and Set-Difference

- These operations take two input relations, which must be *union-compatible*:
  - Two relations R and S are ***union-compatible*** if they have the same number of columns and corresponding columns have the same domains (similar values).

# Example: not union-compatible

(different number of columns)

Anne	aaa	111111
Bob	bbb	222222
Chris	ccc	333333

Tom	1980
Sam	1985
Steve	1986

# Example: not union-compatible

(different domains for the second column)

Anne	aaa
Bob	bbb
Chris	ccc

Tom	1980
Sam	1985
Steve	1986

# Example: union-compatible

Anne	1970
Bob	1971
Chris	1972

Tom	1980
Sam	1985
Steve	1986

# Union

*S1*

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

*S2*

<u>sid</u>	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

$S1 \cup S2$

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0
28	yuppy	9	35.0
44	guppy	5	35.0

# Intersection

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S1

<u>sid</u>	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

S2

<u>sid</u>	sname	rating	age
31	lubber	8	55.5
58	rusty	10	35.0

$S1 \cap S2$

# Set Difference

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S1

<u>sid</u>	sname	rating	age
22	dustin	7	45.0

$S1 - S2$

<u>sid</u>	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

S2

<u>sid</u>	sname	rating	age
28	yuppy	9	35.0
44	guppy	5	35.0

$S2 - S1$

# Cross-Product

- $S1 \times R1$
- Each row of  $S1$  is paired with each row of  $R1$ .
- *Result schema* has one field per field of  $S1$  and  $R1$ , with field names ‘inherited’ if possible.

**$S1$**

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

**$R1$**

<u>sid</u>	<u>bid</u>	<u>day</u>
22	101	10/10/96
58	103	11/12/96

# Cross-Product

S1

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

R1

<u>sid</u>	<u>bid</u>	<u>day</u>
22	101	10/10/96
58	103	11/12/96

# S1 X R1

# Cross-Product

S1

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

R1

<u>sid</u>	<u>bid</u>	<u>day</u>
22	101	10/10/96
58	103	11/12/96

# S1 X R1

# Cross-Product

**S1**

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

**R1**

<u>sid</u>	<u>bid</u>	<u>day</u>
22	101	10/10/96
58	103	11/12/96

**S1 X R1**



(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96

# Cross-Product

**S1**

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

**R1**

<u>sid</u>	<u>bid</u>	<u>day</u>
22	101	10/10/96
58	103	11/12/96

**S1 X R1**



(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96

# Cross-Product

**S1**

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

**R1**

<u>sid</u>	<u>bid</u>	<u>day</u>
22	101	10/10/96
58	103	11/12/96

**S1 X R1**

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96



# Cross-Product

**S1**

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

**R1**

<u>sid</u>	<u>bid</u>	<u>day</u>
22	101	10/10/96
58	103	11/12/96



**S1 X R1**

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96



# Cross-Product

**S1**

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

**R1**

<u>sid</u>	<u>bid</u>	<u>day</u>
22	101	10/10/96
58	103	11/12/96



**S1 X R1**

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	35.0	22	101	10/10/96



# Cross-Product

**S1**

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

**R1**

<u>sid</u>	<u>bid</u>	<u>day</u>
22	101	10/10/96
58	103	11/12/96

**S1 X R1**

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96



Both S1 and R1 have a field called *sid*. Which may cause a conflict when referring to columns

*S1 X R1*

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

# Cross-Product

- $S1 \times R1$ : Each row of  $S1$  paired with each row of  $R1$ .
- Q: How many rows in the result?
- *Result schema* has one field per field of  $S1$  and  $R1$ , with field names 'inherited' if possible.
  - *May have a naming conflict*: Both  $S1$  and  $R1$  have a field with the same name.
  - In this case, can use the *renaming operator*:

$$\rho(C(1 \rightarrow sid1, 5 \rightarrow sid2), S1 \times R1)$$

# Renaming Operator

Takes a relation schema and gives a new name to the schema and the columns

$$\rho (C(1 \rightarrow sid1, 5 \rightarrow sid2), S1 \times R1)$$

C	1 sid1	2 sname	3 rating	4 age	5 sid2	6 bid	7 day
	22	dustin	7	45.0	22	101	10/10/96
	22	dustin	7	45.0	58	103	11/12/96
	31	lubber	8	55.5	22	101	10/10/96
	31	lubber	8	55.5	58	103	11/12/96
	58	rusty	10	35.0	22	101	10/10/96
	58	rusty	10	35.0	58	103	11/12/96

# Compound Operator: Intersection

- In addition to the 5 basic operators, there are several additional “Compound Operators”
  - These add no computational power to the language, but are useful shorthands.
  - Can be expressed solely with the basic ops.
- Intersection takes two input relations, which must be *union-compatible*.
- Q: How to express it using basic operators?

$$R \cap S = R - (R - S)$$

# Compound Operator: Join

- Joins are compound operators involving cross product, selection, and (sometimes) projection.
- Most common type of join is a “*natural join*” (often just called “join”).  $R \bowtie S$  conceptually is:
  - Compute  $R \times S$
  - Select rows where attributes that appear in both relations have equal values
  - Project all unique attributes and one copy of each of the common ones.

# Joins

- Condition Join:  $R \bowtie_C S = \sigma_C(R \times S)$
- *Result schema* same as that of cross-product.
- Fewer tuples than cross-product, might be able to compute more efficiently
- Sometimes called a *theta-join*.

# Joins

*S1*

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

*R1*

<u>sid</u>	<u>bid</u>	<u>day</u>
22	101	10/10/96
58	103	11/12/96

$S1 \bowtie R1$

$S1.sid < R1.sid$

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	58	103	11/12/96

*S1*

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

*R1*

<u>sid</u>	<u>bid</u>	<u>day</u>
22	101	10/10/96
58	103	11/12/96

*S1 X R1*

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

**S1**

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

**R1**

<u>sid</u>	<u>bid</u>	<u>day</u>
22	101	10/10/96
58	103	11/12/96

$$\sigma_{S1.sid < R1.sid}^{(S1 \times R1)}$$

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

*S1*

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

*R1*

<u>sid</u>	<u>bid</u>	<u>day</u>
22	101	10/10/96
58	103	11/12/96

$$\sigma_{S1.sid < R1.sid}^{(S1 \times R1)}$$

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	58	103	11/12/96

Equi-Join: A special case of condition join where the condition  $c$  contains only *equalities*.

$S1$

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

$R1$

<u>sid</u>	<u>bid</u>	<u>day</u>
22	101	10/10/96
58	103	11/12/96

$$S1 \bowtie_{sid} R1$$

sid	sname	rating	age	bid	day

Equi-Join: A special case of condition join where the condition  $c$  contains only *equalities*.

$S1$

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

$R1$

<u>sid</u>	<u>bid</u>	<u>day</u>
22	101	10/10/96
58	103	11/12/96

$$S1 \bowtie_{sid} R1$$

sid	sname	rating	age	bid	day
22	dustin	7	45.0	101	10/10/96

Equi-Join: A special case of condition join where the condition  $c$  contains only *equalities*.

$S1$

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

$R1$

<u>sid</u>	<u>bid</u>	<u>day</u>
22	101	10/10/96
58	103	11/12/96

$$S1 \bowtie_{sid} R1$$

sid	sname	rating	age	bid	day
22	dustin	7	45.0	101	10/10/96

# Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples from one relation that does not match tuples in the other relation to the result of the join.
- Uses *null* values:
  - *null* signifies that the value is unknown or does not exist
  - All comparisons involving *null* are (roughly speaking) **false** by definition.

# Outer Join – Example

- Relation *loan*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

- Relation *borrower*

<i>customer_name</i>	<i>loan_number</i>
Jones	L-170
Smith	L-230
Hayes	L-155

# Outer Join – Example

- Inner Join

*loan*  *Borrower*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

- Left Outer Join

*loan*  *Borrower*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	<i>null</i>

# Outer Join – Example

▫ Right Outer Join

*loan*  *borrower*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
<b>L-155</b>	<i>null</i>	<i>null</i>	<b>Hayes</b>

▫ Full Outer Join

*loan*  *borrower*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
<b>L-260</b>	<b>Perryridge</b>	<b>1700</b>	<i>null</i>
<b>L-155</b>	<i>null</i>	<i>null</i>	<b>Hayes</b>

# Relational Algebra

Basic Operations  
Algebra of Bags

# What is an “Algebra”

- ◆ Mathematical system consisting of:
  - ◆ *Operands* --- variables or values from which new values can be constructed.
  - ◆ *Operators* --- symbols denoting procedures that construct new values from given values.

# What is Relational Algebra?

- ◆ An algebra whose operands are relations or variables that represent relations.
- ◆ Operators are designed to do the most common things that we need to do with relations in a database.
  - ◆ The result is an algebra that can be used as a *query language* for relations.

# Core Relational Algebra

- ◆ Union, intersection, and difference.
  - ◆ Usual set operations, but *both operands must have the same relation schema.*
- ◆ Selection: picking certain rows.
- ◆ Projection: picking certain columns.
- ◆ Products and joins: compositions of relations.
- ◆ Renaming of relations and attributes.

# Selection

◆  $R_1 := \sigma_C(R_2)$

- ◆  $C$  is a condition (as in “if” statements) that refers to attributes of  $R_2$ .
- ◆  $R_1$  is all those tuples of  $R_2$  that satisfy  $C$ .

# Example: Selection

Relation Sells:

bar	beer	price
Joe's	Bud	2.50
Joe's	Miller	2.75
Sue's	Bud	2.50
Sue's	Miller	3.00

JoeMenu :=  $\sigma_{\text{bar}=\text{"Joe's"}}(\text{Sells})$ :

bar	beer	price
Joe's	Bud	2.50
Joe's	Miller	2.75

# Projection

◆  $R_1 := \pi_L(R_2)$

- ◆  $L$  is a list of attributes from the schema of  $R_2$ .
- ◆  $R_1$  is constructed by looking at each tuple of  $R_2$ , extracting the attributes on list  $L$ , in the order specified, and creating from those components a tuple for  $R_1$ .
- ◆ Eliminate duplicate tuples, if any.

# Example: Projection

Relation Sells:

bar	beer	price
Joe's	Bud	2.50
Joe's	Miller	2.75
Sue's	Bud	2.50
Sue's	Miller	3.00

Prices :=  $\pi_{\text{beer}, \text{price}}(\text{Sells})$ :

beer	price
Bud	2.50
Miller	2.75
Miller	3.00

# Extended Projection

- ◆ Using the same  $\pi_L$  operator, we allow the list  $L$  to contain arbitrary expressions involving attributes:
  1. Arithmetic on attributes, e.g.,  $A+B->C$ .
  2. Duplicate occurrences of the same attribute.

# Example: Extended Projection

$$R = ( \begin{array}{|c|c|} \hline A & B \\ \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array} )$$

$$\pi_{A+B \rightarrow C, A, A}(R) =$$

C	A1	A2
3	1	1
7	3	3

# Product

◆  $R_3 := R_1 \times R_2$

- ◆ Pair each tuple  $t_1$  of  $R_1$  with each tuple  $t_2$  of  $R_2$ .
- ◆ Concatenation  $t_1t_2$  is a tuple of  $R_3$ .
- ◆ Schema of  $R_3$  is the attributes of  $R_1$  and then  $R_2$ , in order.
- ◆ But beware attribute  $A$  of the same name in  $R_1$  and  $R_2$ : use  $R_1.A$  and  $R_2.A$ .

# Example: $R3 := R1 \times R2$

$R1($ 

A,	B
1	2
3	4

 $)$

$R2($ 

B,	C
5	6
7	8
9	10

 $)$

$R3($ 

A,	R1.B,	R2.B,	C
1	2	5	6
1	2	7	8
1	2	9	10
3	4	5	6
3	4	7	8
3	4	9	10

 $)$

# Theta-Join

- ◆  $R3 := R1 \bowtie_C R2$ 
  - ◆ Take the product  $R1 \times R2$ .
  - ◆ Then apply  $\sigma_C$  to the result.
- ◆ As for  $\sigma$ ,  $C$  can be any boolean-valued condition.
  - ◆ Historic versions of this operator allowed only  $A \theta B$ , where  $\theta$  is  $=$ ,  $<$ , etc.; hence the name “theta-join.”

# Example: Theta Join

Sells(

	bar,	beer,	price
Joe's	Bud	2.50	
Joe's	Miller	2.75	
Sue's	Bud	2.50	
Sue's	Coors	3.00	

)

Bars(

	name,	addr
Joe's	Maple St.	
Sue's	River Rd.	

)

BarInfo := Sells  $\bowtie_{Sells.bar = Bars.name}$  Bars

BarInfo(

	bar,	beer,	price,	name,	addr
Joe's	Bud	2.50	Joe's	Maple St.	
Joe's	Miller	2.75	Joe's	Maple St.	
Sue's	Bud	2.50	Sue's	River Rd.	
Sue's	Coors	3.00	Sue's	River Rd.	

)

# Natural Join

- ◆ A useful join variant (*natural* join) connects two relations by:
  - ◆ Equating attributes of the same name, and
  - ◆ Projecting out one copy of each pair of equated attributes.
- ◆ Denoted  $R_3 := R_1 \bowtie R_2$ .

# Example: Natural Join

Sells(	bar,	beer,	price	)	Bars(	bar,	addr	)
	Joe's	Bud	2.50			Joe's	Maple St.	
	Joe's	Miller	2.75			Sue's	River Rd.	
	Sue's	Bud	2.50					
	Sue's	Coors	3.00					

BarInfo := Sells  $\bowtie$  Bars

Note: Bars.name has become Bars.bar to make the natural join "work."

BarInfo(	bar,	beer,	price,	addr	)
	Joe's	Bud	2.50	Maple St.	
	Joe's	Miller	2.75	Maple St.	
	Sue's	Bud	2.50	River Rd.	
	Sue's	Coors	3.00	River Rd.	

# Renaming

- ◆ The  $\rho$  operator gives a new schema to a relation.
- ◆  $R1 := \rho_{R1(A1, \dots, An)}(R2)$  makes  $R1$  be a relation with attributes  $A1, \dots, An$  and the same tuples as  $R2$ .
- ◆ Simplified notation:  $R1(A1, \dots, An) := R2$ .

# Example: Renaming

Bars( 

name	addr
Joe's	Maple St.
Sue's	River Rd.

 )

$R(\text{bar}, \text{addr}) := \text{Bars}$

R( 

bar	addr
Joe's	Maple St.
Sue's	River Rd.

 )

# Schemas for Results

- ◆ **Union, intersection, and difference:** the schemas of the two operands must be the same, so use that schema for the result.
- ◆ **Selection:** schema of the result is the same as the schema of the operand.
- ◆ **Projection:** list of attributes tells us the schema.

# Schemas for Results --- (2)

- ◆ **Product**: schema is the attributes of both relations.
  - ◆ Use R.A, etc., to distinguish two attributes named  $A$ .
- ◆ **Theta-join**: same as product.
- ◆ **Natural join**: union of the attributes of the two relations.
- ◆ **Renaming**: the operator tells the schema.

# Relational Algebra on Bags

- ◆ A *bag* (or *multiset*) is like a set, but an element may appear more than once.
- ◆ Example: {1,2,1,3} is a bag.
- ◆ Example: {1,2,3} is also a bag that happens to be a set.

# Why Bags?

- ◆ SQL, the most important query language for relational databases, is actually a bag language.
- ◆ Some operations, like projection, are more efficient on bags than sets.

# Operations on Bags

- ◆ **Selection** applies to each tuple, so its effect on bags is like its effect on sets.
- ◆ **Projection** also applies to each tuple, but as a bag operator, we do not eliminate duplicates.
- ◆ **Products** and **joins** are done on each pair of tuples, so duplicates in bags have no effect on how we operate.

# Example: Bag Selection

$$R( \begin{array}{|c|c|} \hline A, & B \\ \hline 1 & 2 \\ 5 & 6 \\ 1 & 2 \\ \hline \end{array} )$$

$$\sigma_{A+B < 5}(R) = \begin{array}{|c|c|} \hline A & B \\ \hline 1 & 2 \\ 1 & 2 \\ \hline \end{array}$$

# Example: Bag Projection

$$R( \begin{array}{|c|c|} \hline A, & B \\ \hline 1 & 2 \\ 5 & 6 \\ 1 & 2 \\ \hline \end{array} )$$

$$\pi_A(R) = \begin{array}{|c|} \hline A \\ \hline 1 \\ 5 \\ 1 \\ \hline \end{array}$$

# Example: Bag Product

$$R( \begin{array}{|c|c|} \hline A, & B \\ \hline 1 & 2 \\ 5 & 6 \\ 1 & 2 \\ \hline \end{array} )$$
$$S( \begin{array}{|c|c|} \hline B, & C \\ \hline 3 & 4 \\ 7 & 8 \\ \hline \end{array} )$$
$$R \times S = \begin{array}{|c|c|c|c|} \hline A & R.B & S.B & C \\ \hline 1 & 2 & 3 & 4 \\ 1 & 2 & 7 & 8 \\ 5 & 6 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 7 & 8 \\ \hline \end{array}$$

# Example: Bag Theta-Join

$$R( \begin{array}{|c|c|} \hline A, & B \\ \hline \end{array} )$$

A,	B
1	2
5	6
1	2

$$S( \begin{array}{|c|c|} \hline B, & C \\ \hline \end{array} )$$

B,	C
3	4
7	8

$$R \bowtie_{R.B < S.B} S =$$

A	R.B	S.B	C
1	2	3	4
1	2	7	8
5	6	7	8
1	2	3	4
1	2	7	8

# Bag Union

- ◆ An element appears in the union of two bags the sum of the number of times it appears in each bag.
- ◆ Example:  $\{1,2,1\} \cup \{1,1,2,3,1\} = \{1,1,1,1,1,2,2,3\}$

# Bag Intersection

- ◆ An element appears in the intersection of two bags the minimum of the number of times it appears in either.
- ◆ **Example:**  $\{1,2,1,1\} \cap \{1,2,1,3\} = \{1,1,2\}$ .

# Bag Difference

- ◆ An element appears in the difference  $A - B$  of bags as many times as it appears in  $A$ , minus the number of times it appears in  $B$ .
  - ◆ But never less than 0 times.
- ◆ Example:  $\{1,2,1,1\} - \{1,2,3\} = \{1,1\}$ .

# Banking Example

- branch (branch\_name, branch\_city, assets)
- customer (customer\_name, customer\_street, customer\_city)
- account (account\_number, branch\_name, balance)
- loan (loan\_number, branch\_name, amount)
- depositor (customer\_name, account\_number)
- borrower (customer\_name, loan\_number)

# Write the following Queries in RA.

- branch (branch\_name, branch\_city, assets)
- customer (customer\_name, customer\_street, customer\_city)
- account (account\_number, branch\_name, balance)
- loan (loan\_number, branch\_name, amount)
- depositor (customer\_name, account\_number)
- borrower (customer\_name, loan\_number)

1. Find all loans of over \$1200
2. Find the loan number for each loan of an amount greater than \$1200
3. Find the names of all customers who have a loan, an account, or both, from the bank
4. Find the names of all customers who have a loan and an account at the bank

# Write the following Queries in RA.

- branch (branch\_name, branch\_city, assets)
- customer (customer\_name, customer\_street, customer\_city)
- account (account\_number, branch\_name, balance)
- loan (loan\_number, branch\_name, amount)
- depositor (customer\_name, account\_number)
- borrower (customer\_name, loan\_number)

5. Find the names of all customers who have a loan at the Perryridge branch
6. Find the names of all customers who have a loan at the Perryridge branch but do not have an account at any branch of the bank
7. Find the names of all customers who have a loan at the Perryridge branch
8. Find the largest account balance

1 & 2

$\sigma_{\text{amount} > 1200}(\text{loan})$

$\prod_{\text{loan\_number}} (\sigma_{\text{amount} > 1200}(\text{loan}))$

3 & 4

$$\Pi_{\text{customer\_name}}(\text{borrower}) \cup \Pi_{\text{customer\_name}}(\text{depositor})$$

$$\Pi_{\text{customer\_name}(\text{borrower})} \cap \Pi_{\text{customer\_name}}(\text{depositor})$$

5 Find the names of all customers who have a loan at the Perryridge branch

$$\prod_{\text{customer\_name}} (\sigma_{\text{branch\_name}=\text{"Perryridge"}} (\sigma_{\text{borrower.loan\_number} = \text{loan.loan\_number}} (\text{borrower} \times \text{loan})))$$

6 Find the names of all customers who have a loan at the Perryridge branch but do not have an account at any branch of the bank

$$\begin{aligned} & \prod_{\text{customer\_name}} (\sigma_{\text{branch\_name} = \text{"Perryridge"}} ( \\ & \sigma_{\text{borrower.loan\_number} = \text{loan.loan\_number}} (\text{borrower} \times \text{loan}))) \\ & - \prod_{\text{customer\_name}} (\text{depositor}) \end{aligned}$$

7 Find the names of all customers who have a loan at the Perryridge branch

■ Answer 1

$$\prod_{customer\_name} (\sigma_{branch\_name = "Perryridge"} ( \sigma_{borrower.loan\_number = loan.loan\_number} (borrower \times loan)))$$

■ Answer 2

$$\prod_{customer\_name} (\sigma_{loan.loan\_number = borrower.loan\_number} ( (\sigma_{branch\_name = "Perryridge"} (loan)) \times borrower))$$

**8** (Aggregate max is not directly supported in relational algebra – Find those balances that are not the largest • Rename account relation as d so that we can compare each account balance with all the others – Use set difference to find the max balance accounts)

$$\blacksquare \Pi_{\text{balance}}(\text{account}) - \Pi_{\text{balance}}(\text{account}) \\ (\sigma_{\text{account.balance} < \text{d.balance}} (\text{account} \times \rho_d(\text{account})))$$

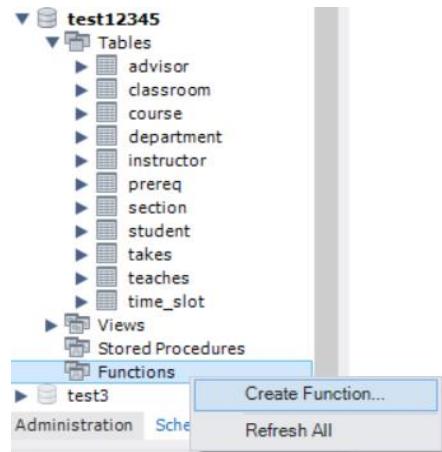
## Functions and Stored Procedures Practice Sheet

Run this command first in SQL qindow:

```
Set global log_bin_trust_function_creators = 1;
```

Practice 1. Create a function to find the number of courses taught by an instructor ID

**Create Function**



**Function Code**

A screenshot of the MySQL Workbench code editor. The 'Name:' field contains 'new\_function'. The 'DDL:' tab shows the following SQL code:

```
1 • CREATE FUNCTION getCourseCount (Instructor_ID varchar(5))
2   RETURNS varchar(30)
3   BEGIN
4     Declare CorseCount varchar(30);
5     select count(course_id) into CorseCount from teaches where id = Instructor_ID;
6     RETURN CorseCount;
7   END
8
```

The code editor has a toolbar with various icons. At the bottom right, there are 'Apply' and 'Revert' buttons, with 'Apply' being circled in red.

```
Set global log_bin_trust_function_creators = 1;
```

**Call function**

A screenshot of the MySQL Workbench SQL editor. The tabs at the top show 'SQL File 5\*', 'SQL File 10\*', 'SQL File 6\*', 'SQL File 7\*', 'SQL File 8\*', 'SQL File 9\*', and 'SQL File 9\*'. The current tab is 'SQL File 9\*'. The query pane contains:

```
1 • select getCourseCount('76766') from dual;
```

The result pane shows the output of the query: '1'.

## **Practice 2. Create Stored Procedure (SP) to select data from a table (get all courses)**

### **1. Create SP**

The screenshot shows a database interface with a tree view of schemas. The 'test12345' schema is expanded, showing its tables: advisor, classroom, course, department, instructor, prereq, section, student, takes, teaches, and time\_slot. Below the tables is a 'Views' node. Under 'test12345' is a 'Stored Procedures' node, which is currently selected and highlighted in blue. A context menu is open over this node, with the option 'Create Stored Procedure...' highlighted in blue. Other options in the menu include 'Functions' and 'test3'. At the bottom of the interface, there are tabs for 'Administration' and 'Schemas', with 'Schemas' being the active tab.

### **2. Write Code**

Name:

DDL:

```
1 • CREATE PROCEDURE getCourseInfo()
2   BEGIN
3     select * from course;
4   END
```

### **3. Call SP**

```
1 • call getCourseInfo();
```

### **Practice 3. Create Stored Procedure (SP) to create a NEW table of your choice**

Hint: like previous SP, just replace the select statement with create statement.

### **Practice 4. Create SP to get the average salary of the department of a given instructor**

Code of SP:

Name: `getDeptAVGSAL` The name of the routine is parsed automatically from the DDL statement. The DDL is parsed automatically while you type.

DDL:

```
1 • CREATE PROCEDURE getDeptAVGSAL(IN Inst_ID varchar(20), out DEP_AVG_SAL numeric(6.2))
2 • BEGIN
3 •     select avg(i.salary) into DEP_AVG_SAL from instructor i
4 •     where i.dept_name = (select j.dept_name from instructor j where j.id = Inst_ID) ;
5 • END
```

Call SP:

SQL File 9\*   SQL File 9\*   instructor   SQL File 25\*   course   takes

1 • **call** getDeptAVGSAL('12121',@avgsal);
2 • **select** @avgsal **from** dual



# Java Database Connectivity (JDBC)



# Java Database Connectivity (JDBC)

1. Connection.
2. Statement.
3. Process Result (Select).
4. Close.



# Java Database Connectivity (JDBC)

## 1. Connection:

- ◊ Register Driver (`mysql-connector.jar`) → [Download MySQL Connector/J](#)
- ◊ Connect
  - URL: `jdbc:mysql://server-name:port/database-name`
  - Username
  - Password



# Java Database Connectivity (JDBC)

## 2. Statement (select, insert, update, delete, create, alter, drop, ....)

### ◆ Prepared Statement:

- executeQuery → Result set
- executeUpdate → int
- execute → Boolean (true: Result set, false: int)

### ◆ Callable Statement (procedure, function, package)



# Java Database Connectivity (JDBC)

3. Process result (select)
4. Close connection



# JDBC → Connection

```
// Check Driver
try {
    Class.forName("com.mysql.cj.jdbc.Driver");
    System.out.println("MySQL driver found");
} catch (ClassNotFoundException e) {
    System.out.println("MySQL driver not found");
}
```

```
// Connect to Database
try {
    Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/scott",
                                                "root",
                                                "root");

    System.out.println("Connected to MySQL");
    conn.close();
} catch (SQLException e) {
    System.out.println("Not connected to MySQL");
}
```



# JDBC → Select

```
PreparedStatement ps = conn.prepareStatement("select empno, ename, sal, job from emp");
ResultSet rs = ps.executeQuery();
while (rs.next()) {
    System.out.println(rs.getInt("empno") + " " + rs.getString("ename") + " " + rs.getDouble("sal"));
}

rs.close();
ps.close();
conn.close();
```



# JDBC → Insert

```
Connection conn = null;
PreparedStatement ps = null;
try {
    conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/scott", "root", "root");
    ps = conn.prepareStatement("insert into emp(empno, ename, sal, deptno) values(?, ?, ?, ?)");
    ps.setInt(1, 7777);
    ps.setString(2, "Sami");
    ps.setDouble(3, 1000.0);
    ps.setInt(4, 10);

    ps.executeUpdate();

} catch (SQLException e) {
    e.printStackTrace();
} finally {
    ps.close();
    conn.close();
}
```

# THANKS!

ANY QUESTIONS?

