



ICT 3157: DATABASE SYSTEMS

1. Database and database users
Girija Attigeri

Contents

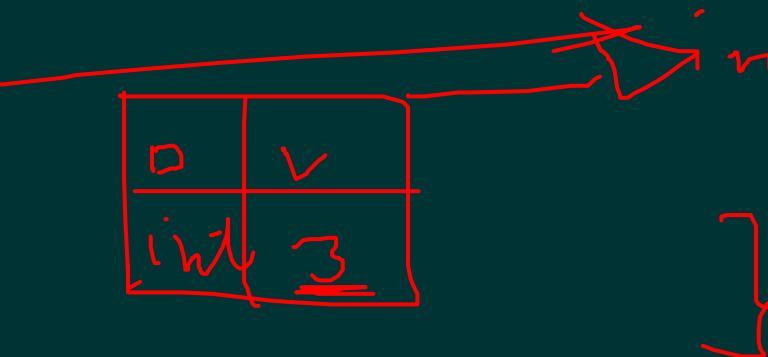
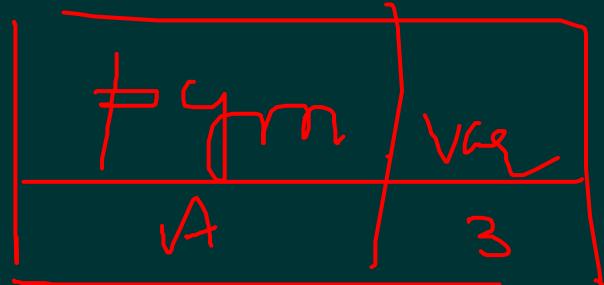
- Introduction to the relational database
- View of data
- Advantages of using a DBMS
- Actors on the scene
- Database architecture

Interaction with database

- Bank
- Hotel, Airline reservation
- Library catalog search
- Purchasing items

Complexity

- Traditional database applications
- Multimedia databases
- Geographic information systems
- Data warehouses and Online analytical processing
- Realtime and active database technology



Met a D a t a

Typn A

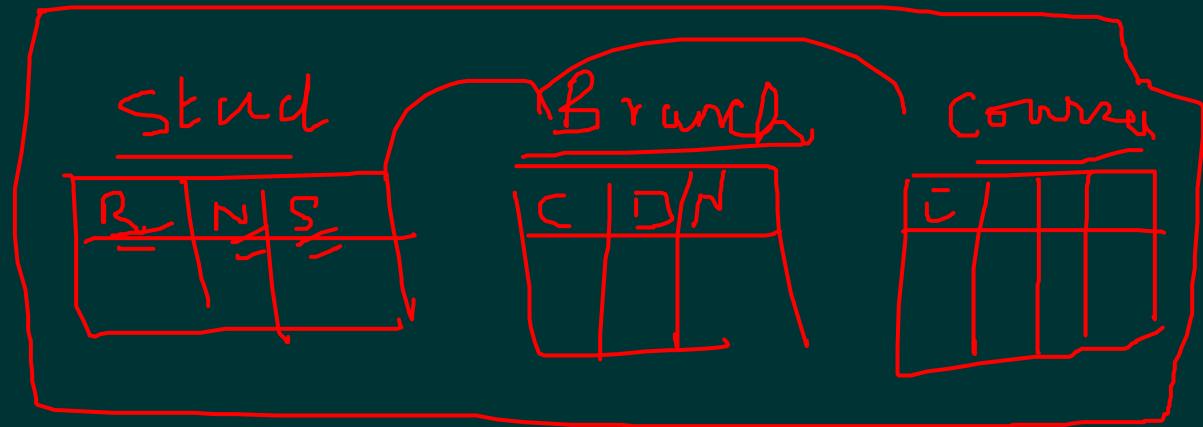
{

int a=10, b=20

int c = a+b;
print(c);

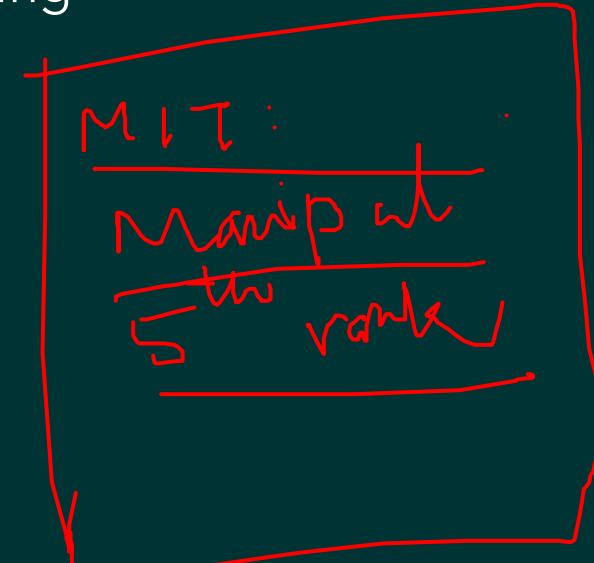
Database

- Collection of **related data**
- Data:
 - Known facts that can be recorded and have implicit meaning



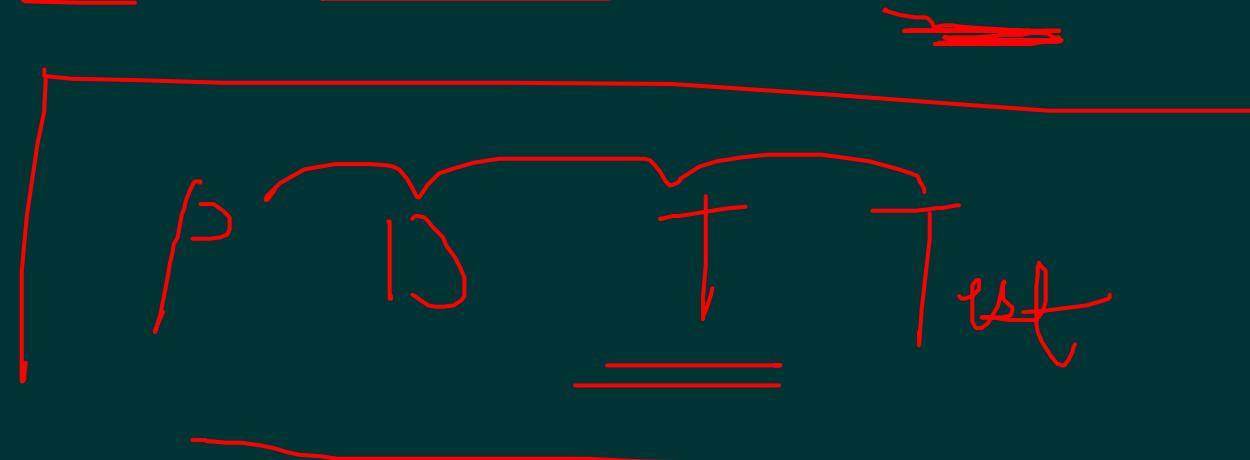
A hand-drawn diagram illustrating a database schema. It features two tables: 'temp' and 'City'. The 'temp' table has two rows, each containing a single value: '25,' and '23,'. The 'City' table has two rows, each containing two values: 'Manifa' and 'Bang' respectively. Arrows point from the 'temp' table to the 'City' table, indicating a relationship between them.

25,	Manifa
23,	Bang



Implicit properties of database

- A database represents some aspect of the real world (Miniworld, UoD)
- Logically coherent collection of data with some inherent meaning
- It is designed, built and populated with data for a specific purpose



Database Management System (DBMS)

- Collection of programs to create, maintain, and access the data
- An environment that is both convenient and efficient to use

SW



Terminologies:

- Defining ✓
- Meta-data ✓
- Constructing ✓
- Manipulating ✓
- Sharing ✓

- UT
- Application programs
 - Query
 - Transaction
 - Manipulating
 - Sharing

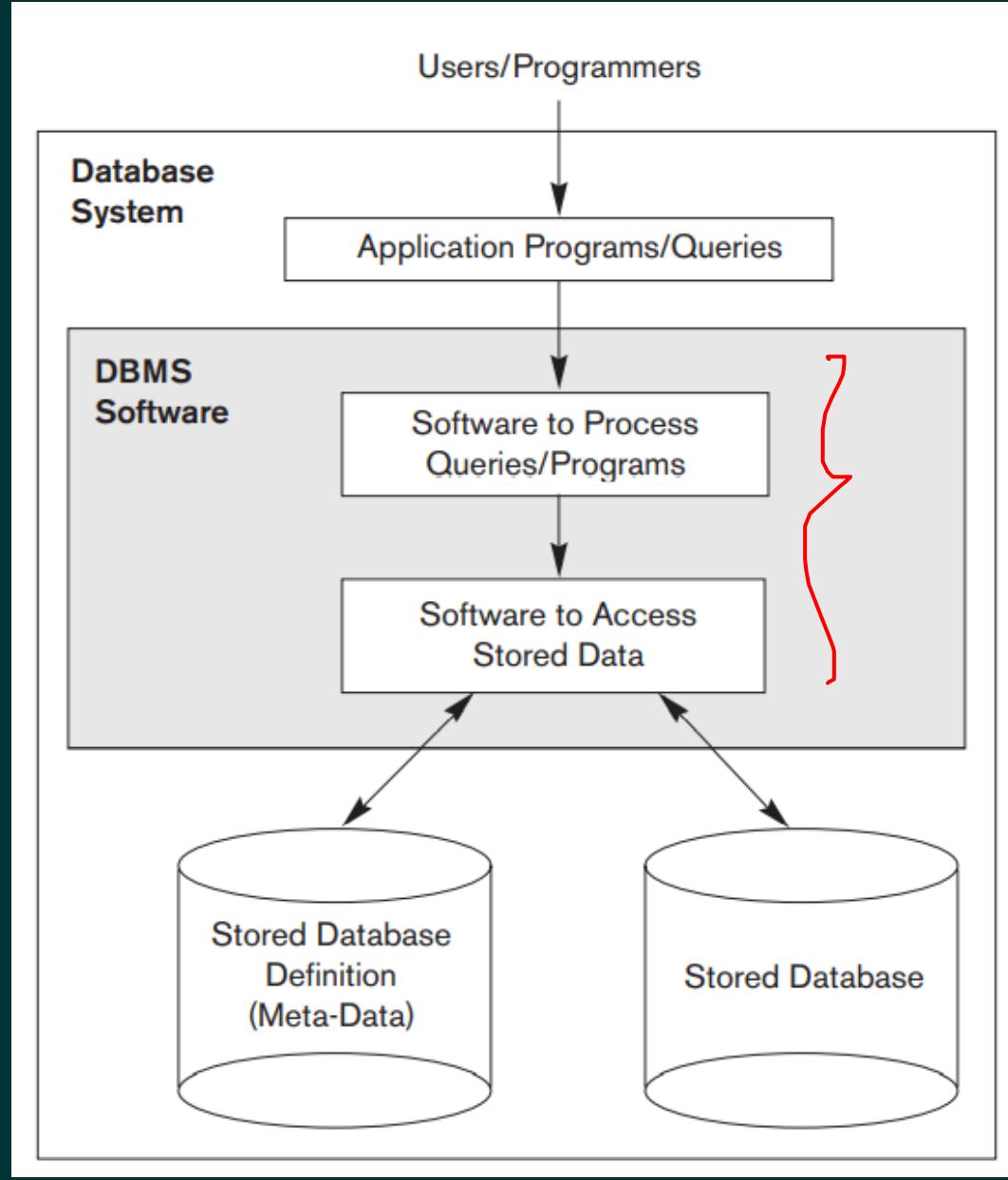
Functionalities:

- Protection
- System protection
- Security
- Maintenance



DATABASE SYSTEMS: Database + DBMS Software

Simplified view of
database system
environment



STUDENT

Name	Student_number	Class	Major
Smith	17	1	CS
Brown	8	2	CS

COURSE

Course_name	Course_number	Credit_hours	Department
Intro to Computer Science	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Database	CS3380	3	CS

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	07	King
92	CS1310	Fall	07	Anderson
102	CS3320	Spring	08	Knuth
112	MATH2410	Fall	08	Chang
119	CS1310	Fall	08	Anderson
135	CS3380	Fall	08	Stone

GRADE_REPORT

Student_number	Section_identifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

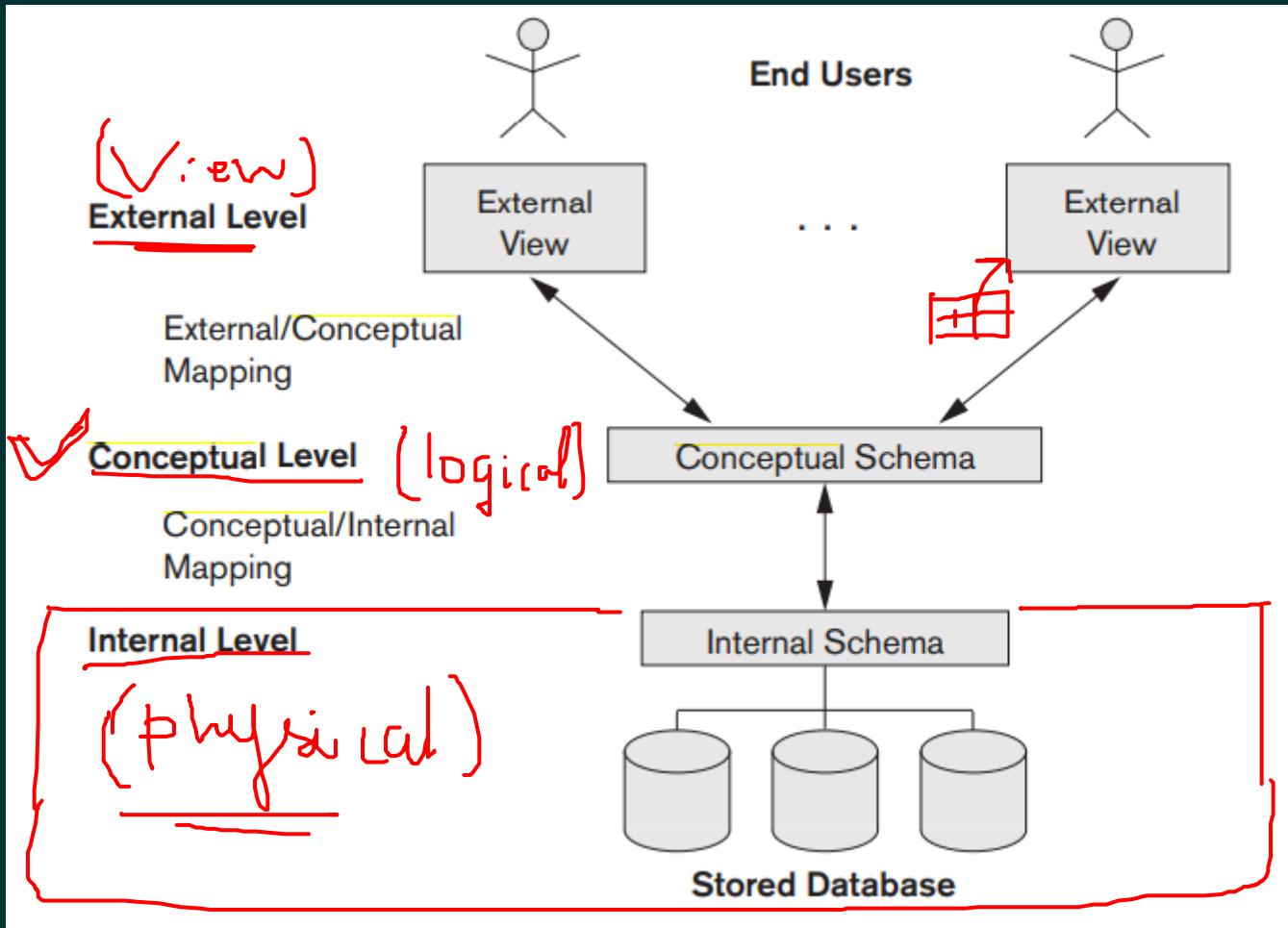
RELATIONS

Relation_name	No_of_columns
STUDENT	4
COURSE	4
SECTION	5
GRADE_REPORT	3
PREREQUISITE	2

COLUMNS

Column_name	Data_type	Belongs_to_relation
Name	Character (30)	STUDENT
Student_number	Character (4)	STUDENT
Class	Integer (1)	STUDENT
Major	Major_type	STUDENT
Course_name	Character (10)	COURSE
Course_number	XXXXNNNN	COURSE
....
....
....
Prerequisite_number	XXXXNNNN	PREREQUISITE

The three-schema architecture (Views).



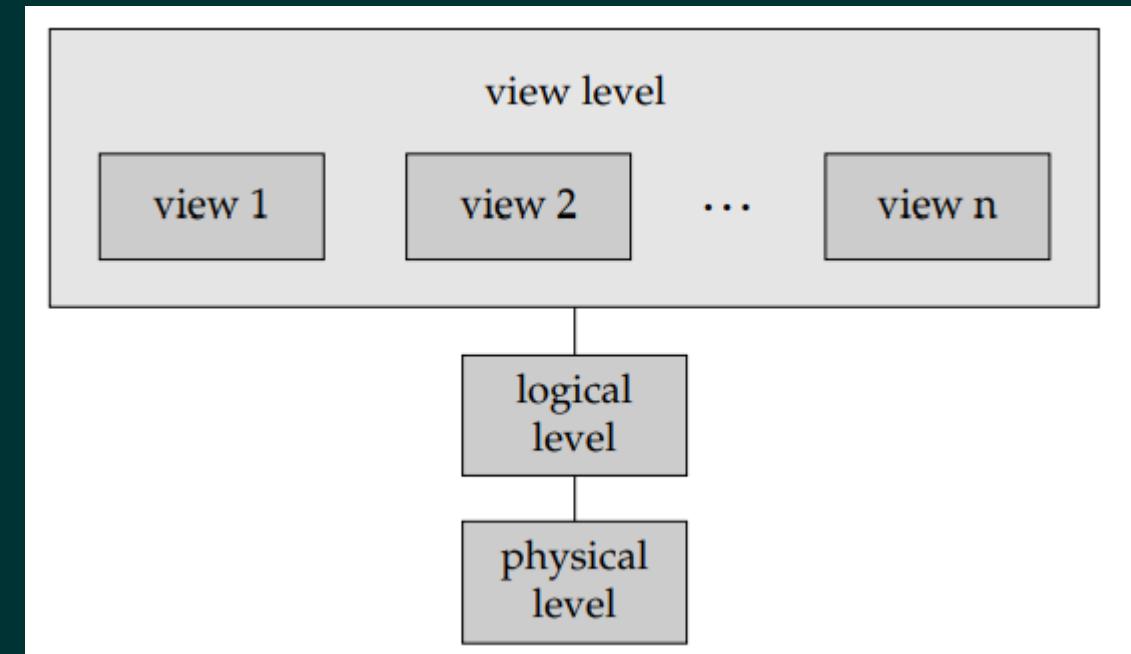
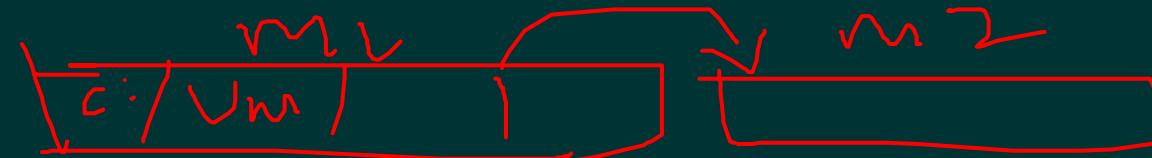
{ Navarie

{ Korth }

View of the data: to provide data abstractions

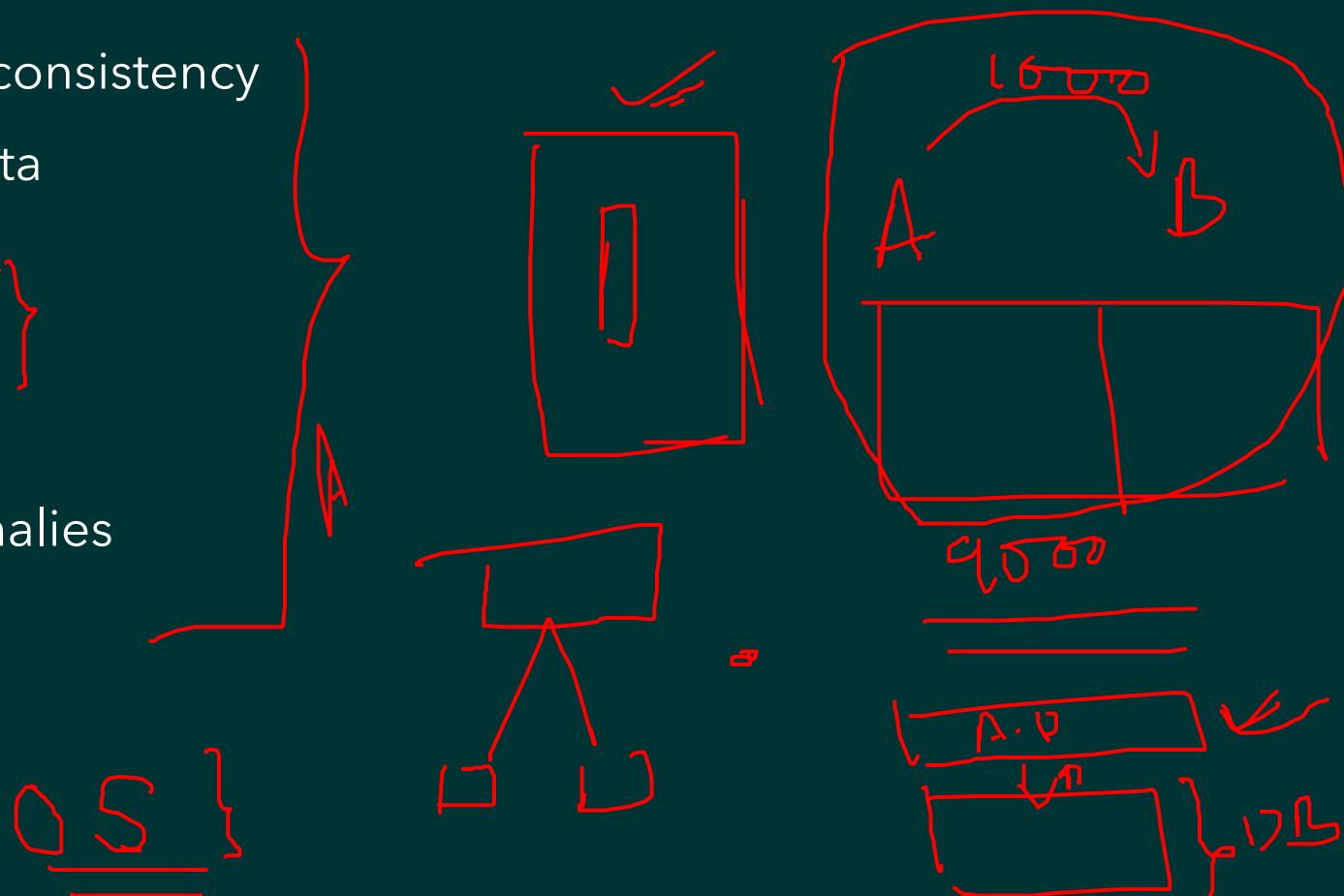
- Physical level (Internal) :
 - Physical data model, data storage, access path
- Logical level (Conceptual):
 - Describes entire database: entities, data types, relationships, constraints...
 - Physical data independence
- View Level (External):
 - Describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group

logical



Database Systems versus File Systems

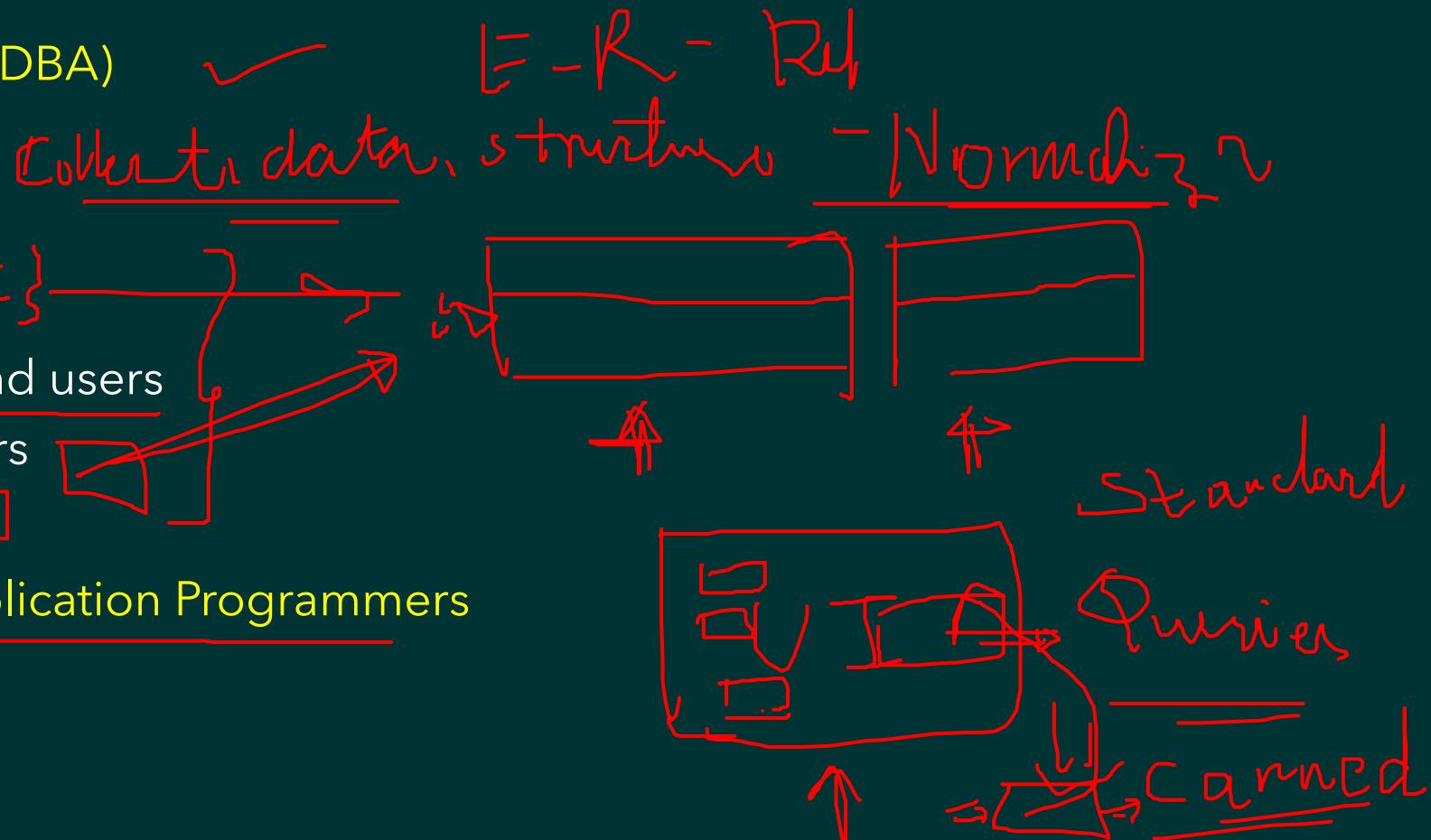
- Data redundancy and inconsistency
- Difficulty in accessing data
- Data isolation
- Integrity problems
- Atomicity problems.
- Concurrent-access anomalies
- Security problems



Actors on the Scene

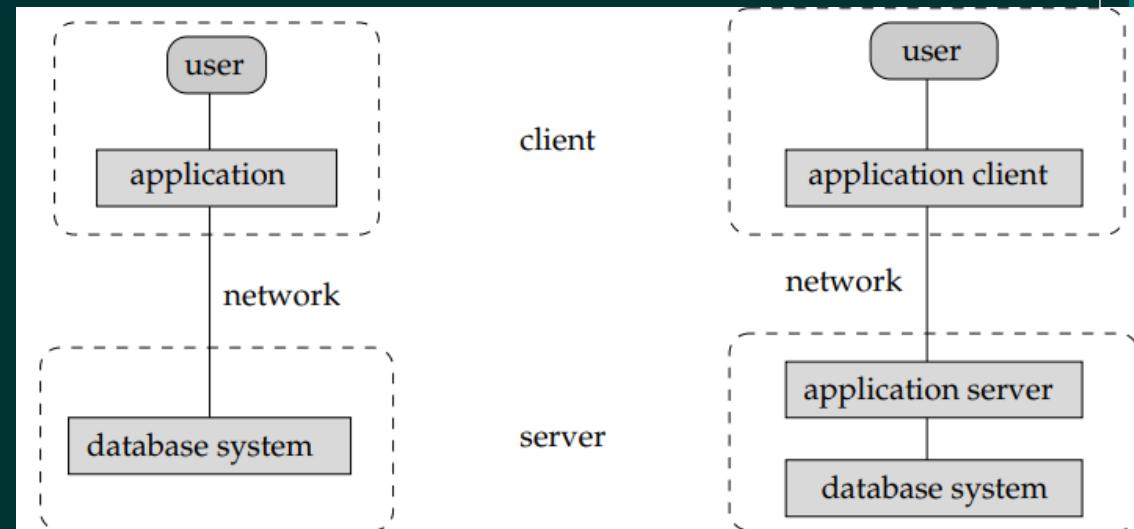
- Database Administrator (DBA) ✓
- Database Designer : Collects data, structures
- End Users
 - Casual end users
 - Naive or parametric end users
 - Sophisticated end users
 - Standalone users :
 - System Analysts and Application Programmers

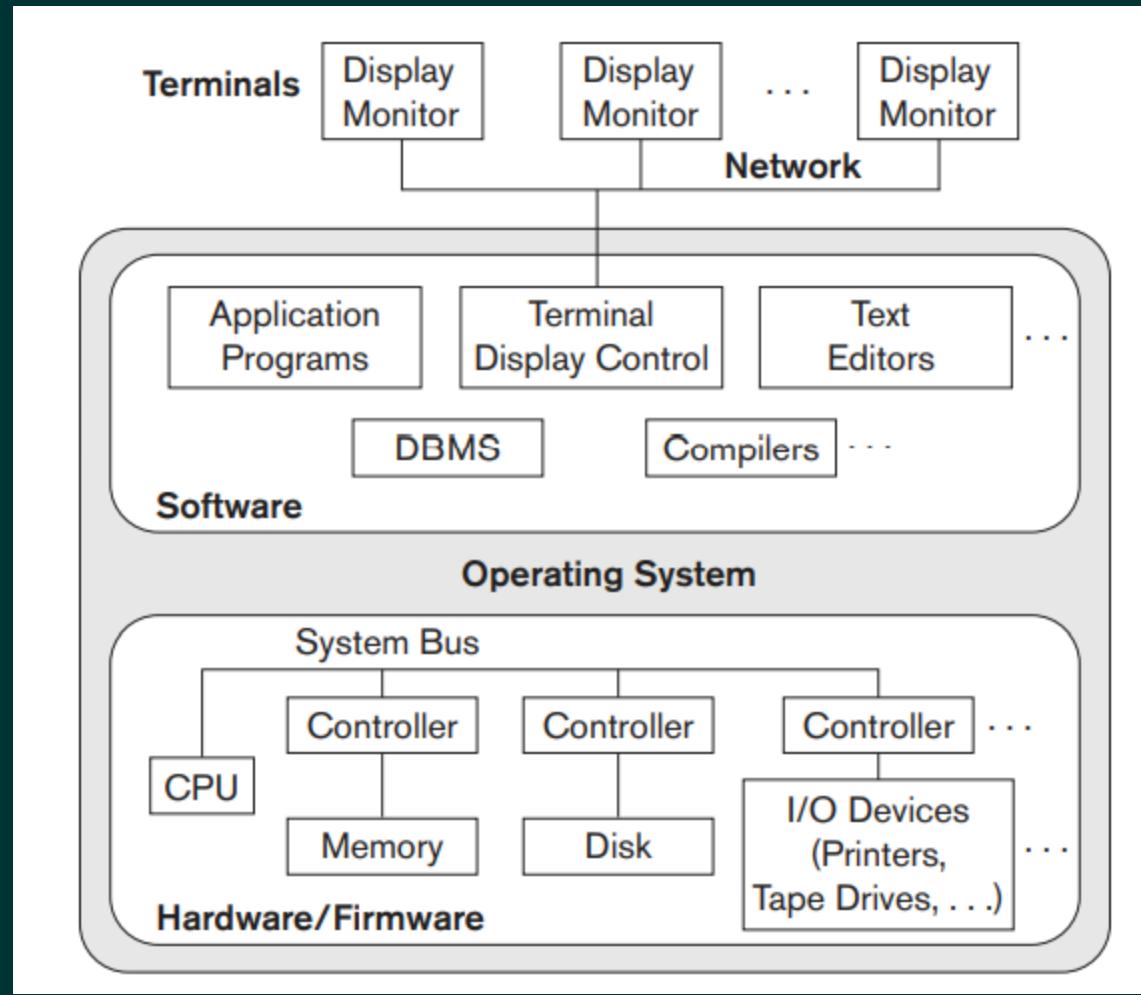
◦ Specialized

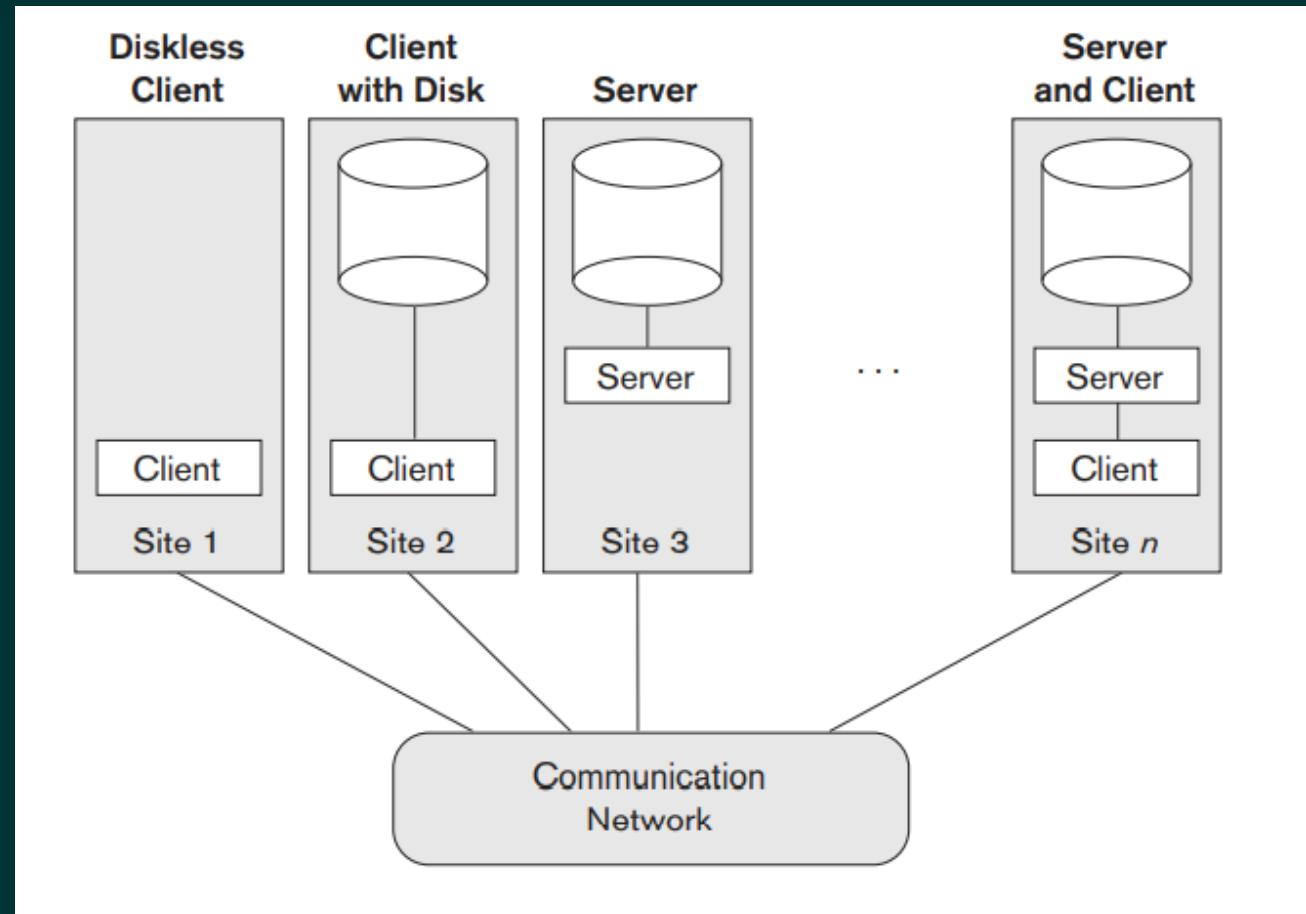
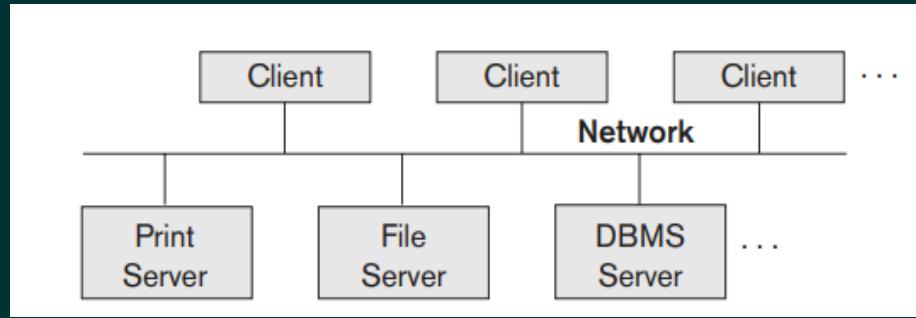


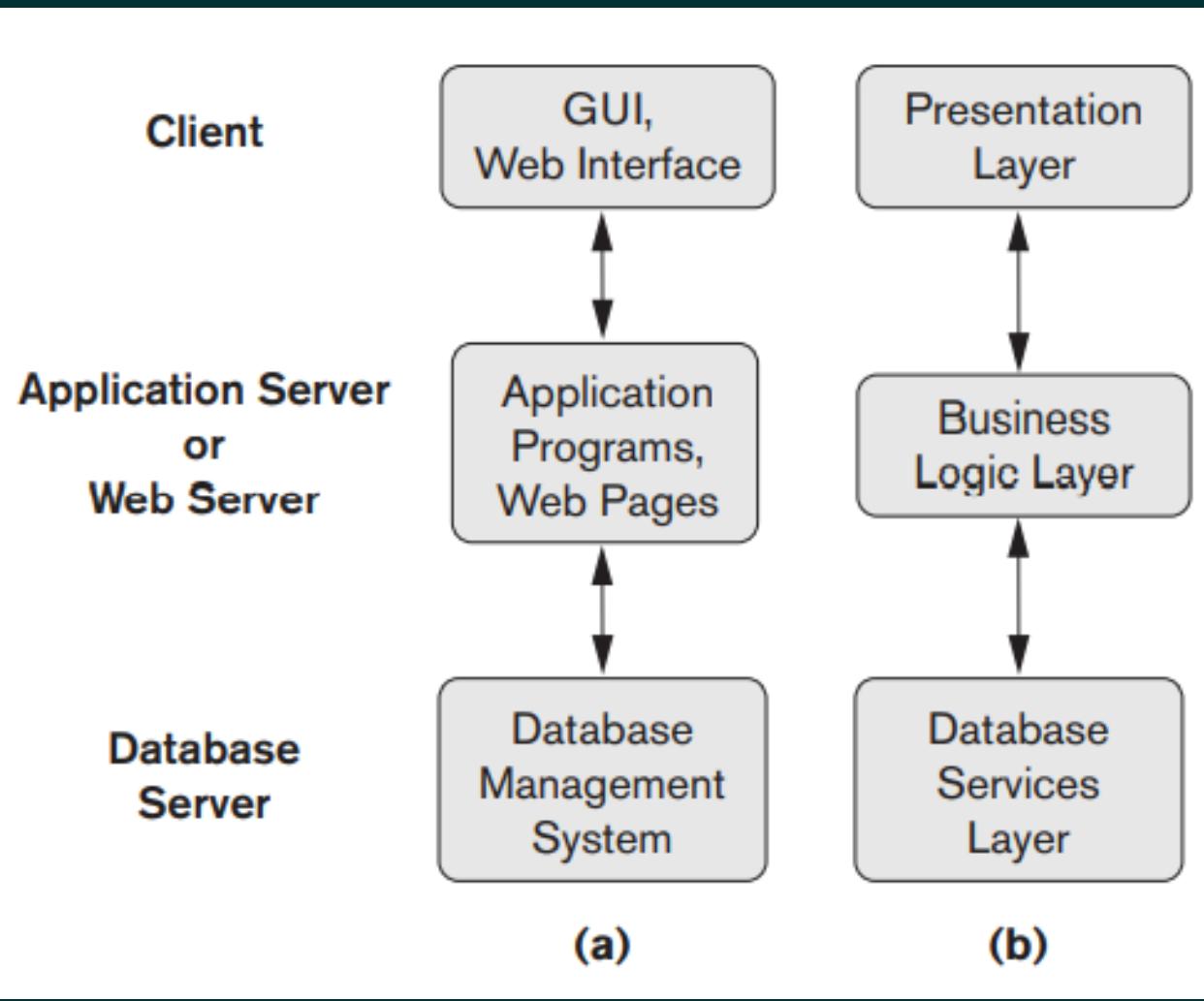
Database Architectures

- Centralized
 - located and stored in a single location
- Client Server
 - Users connect to the database through network
 - Two tier architecture
 - Three tier architecture
- Parallel
 - multiple processors work in parallel on the database to provide the services.
- Distributed
 - collection of multiple interconnected databases, which are spread physically across various locations that communicate via a computer network.



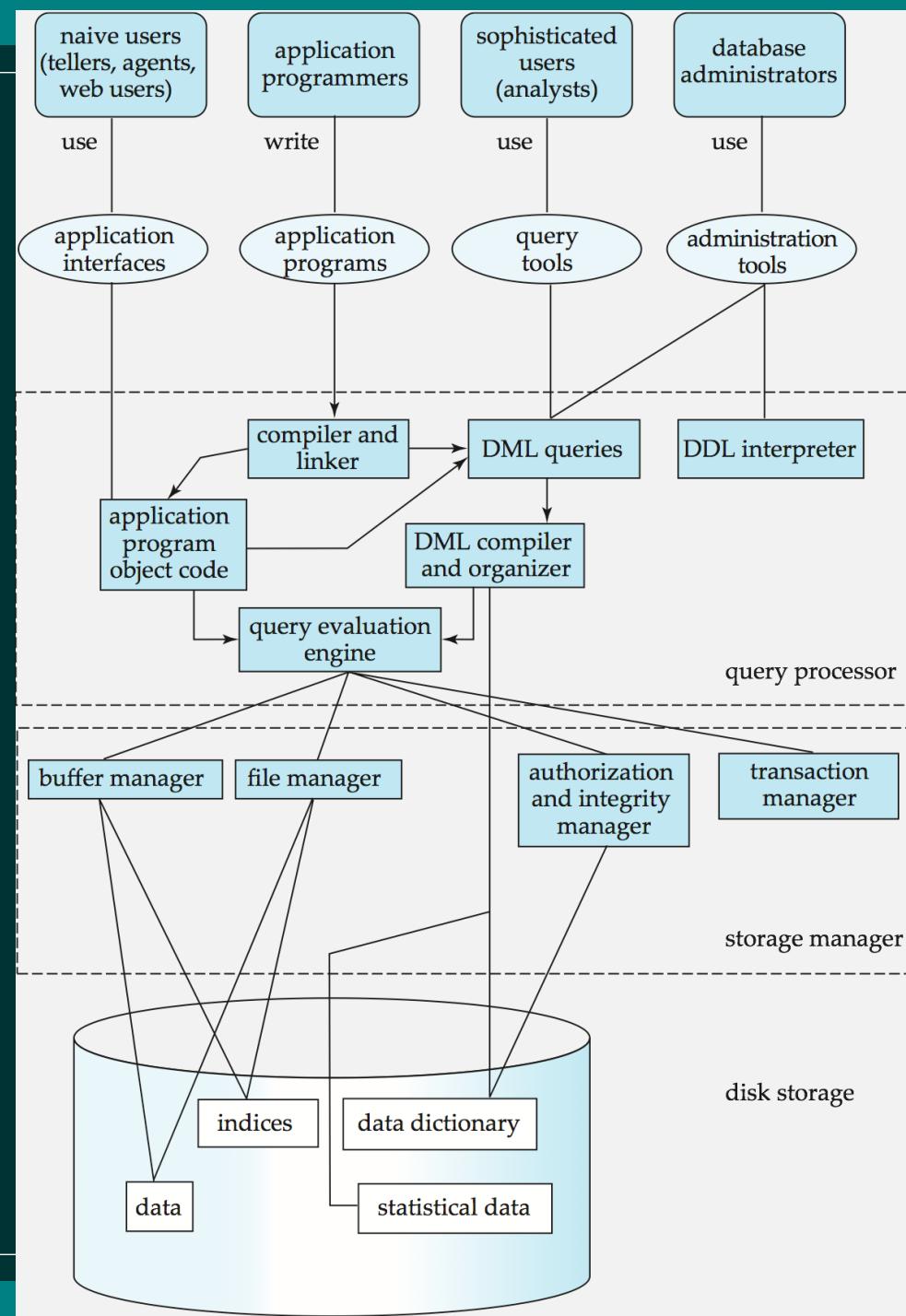






Advantages of Using the DBMS Approach

- Controlling Redundancy
- Restricting Unauthorized Access
- Providing Persistent Storage for program objects
- Providing Storage Structures and Search Techniques for Efficient Query Processing
- Providing Backup and Recovery
- Providing Multiple User Interfaces
- Representing Complex Relationships among Data
- Enforcing Integrity Constraints
- Permitting Inferencing and Actions Using Rules and Triggers



Schemas, Instances, and Database State

- The description of a database is called the database schema, which is specified during database design and is not expected to change frequently

STUDENT			
Name	Student_number	Class	Major
COURSE			
Course_name	Course_number	Credit_hours	Department
PREREQUISITE			
Course_number	Prerequisite_number		
SECTION			
Section_identifier	Course_number	Semester	Year
GRADE_REPORT			
Student_number	Section_identifier	Grade	

Schemas, Instances, and Database State

- Instance - the actual content of the database at a particular point in time
 - Analogous to the value of a variable

STUDENT

Name	Student_number	Class	Major
Smith	17	1	CS
Brown	8	2	CS

COURSE

Course_name	Course_number	Credit_hours	Department
Intro to Computer Science	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Database	CS3380	3	CS

Schemas, Instances, and Database State

- The data in the database at a particular moment in time is called a **database state** or **snapshot**



ICT 3157: DATABASE SYSTEMS

2. Relational Databases

Girija Attigeri

Contents

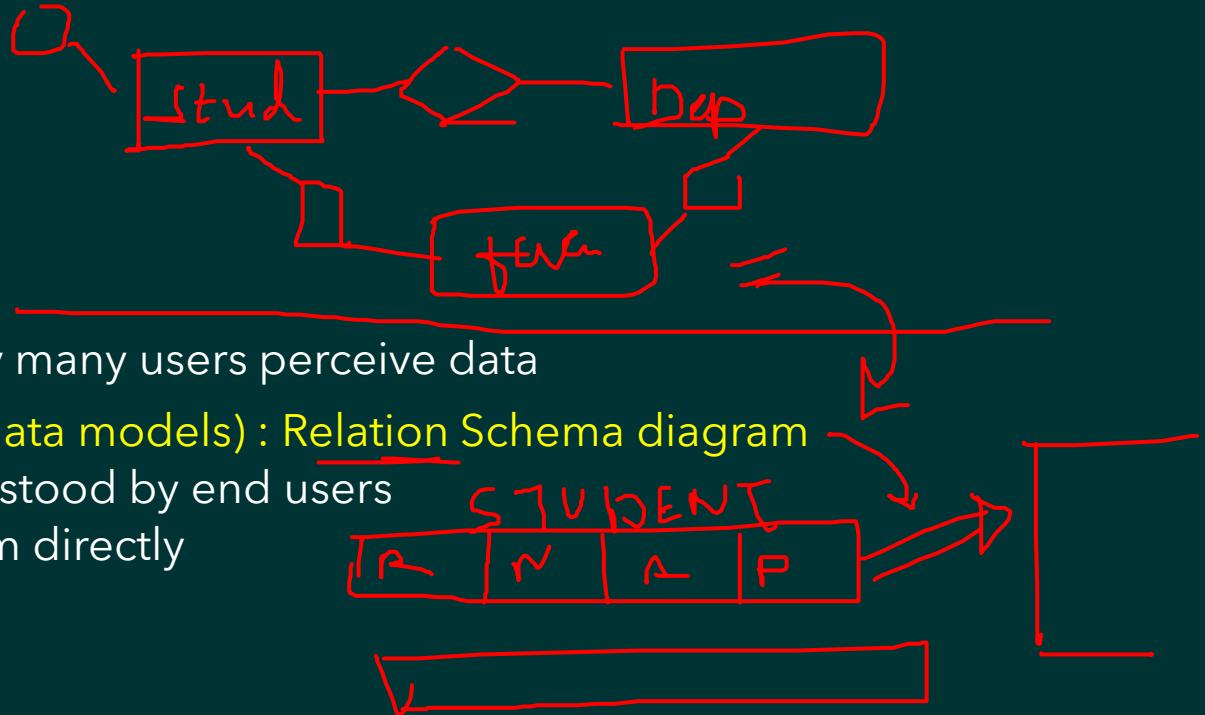
- Introduction to relational model
- Structure of Relational Databases
- Database Schema
- Keys
- Schema Diagrams
- Relational Query Languages
- Relational algebra

Data Model

- Collection of concepts for describing structure of the database
 - Data
 - Relationships
 - Constraints

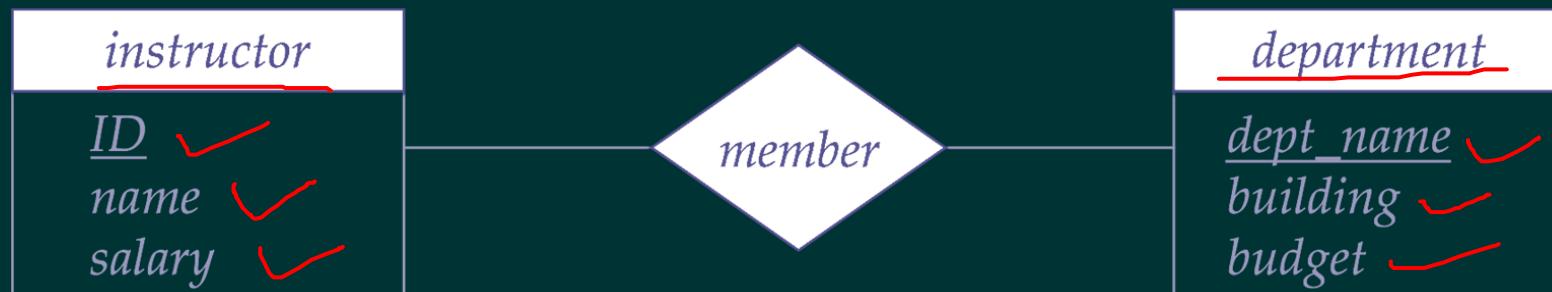
Data Model

- Conceptual data model : E-R Diagram ✓
 - provide concepts that are close to the way many users perceive data
- Representational data model (record based data models) : Relation Schema diagram
 - provide concepts that may be easily understood by end users
 - can be implemented on a computer system directly
- Object data model : ODMG
 - Conceptual data model
- Self-describing data models : XML, Key-Value Pair, NoSQL
 - Data storage using these model combines the description of the data with the data values themselves.
- Physical data models
 - Provide concepts that describe the details of how data is stored on the computer storage media.
 - Generally meant for computer specialists, not for end users
- Older data models: Network, hierarchical



Data Model: Conceptual data model

- The Entity-Relationship Model
- Entities, attributes, and relationships
- Entity: Represents a real-world object or concept
- Attribute: Represents some property of interest that further describes an entity
- Relationships: Relationship among two or more entities represents an association among the entities



Data Model: Conceptual data model

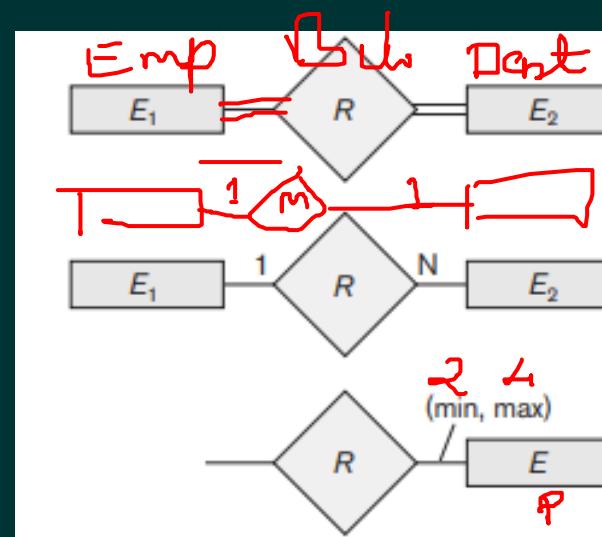


Entity

- The company is organized into departments. Each department has a unique name, a unique number, and a particular employee who manages the department. We keep track of the start date when that employee began managing the department. A department may have several locations.
- A department controls a number of projects, each of which has a unique name, a unique number, and a single location.
- The database will store each employee's name, Social Security number, address, salary, gender, and birth date. An employee is assigned to one department, but may work on several projects, which are not necessarily controlled by the same department. It is required to keep track of the current number of hours per week that an employee works on each project, as well as the direct supervisor of each employee (who is another employee).
- The database will keep track of the dependents of each employee for insurance purposes, including each dependent's first name, sex, birth date, and relationship to the employee.



Symbol	Meaning
	Entity
	Weak Entity
	Relationship
	Identifying Relationship
	Attribute
	Key Attribute
	Multivalued Attribute
	Composite Attribute
	Derived Attribute

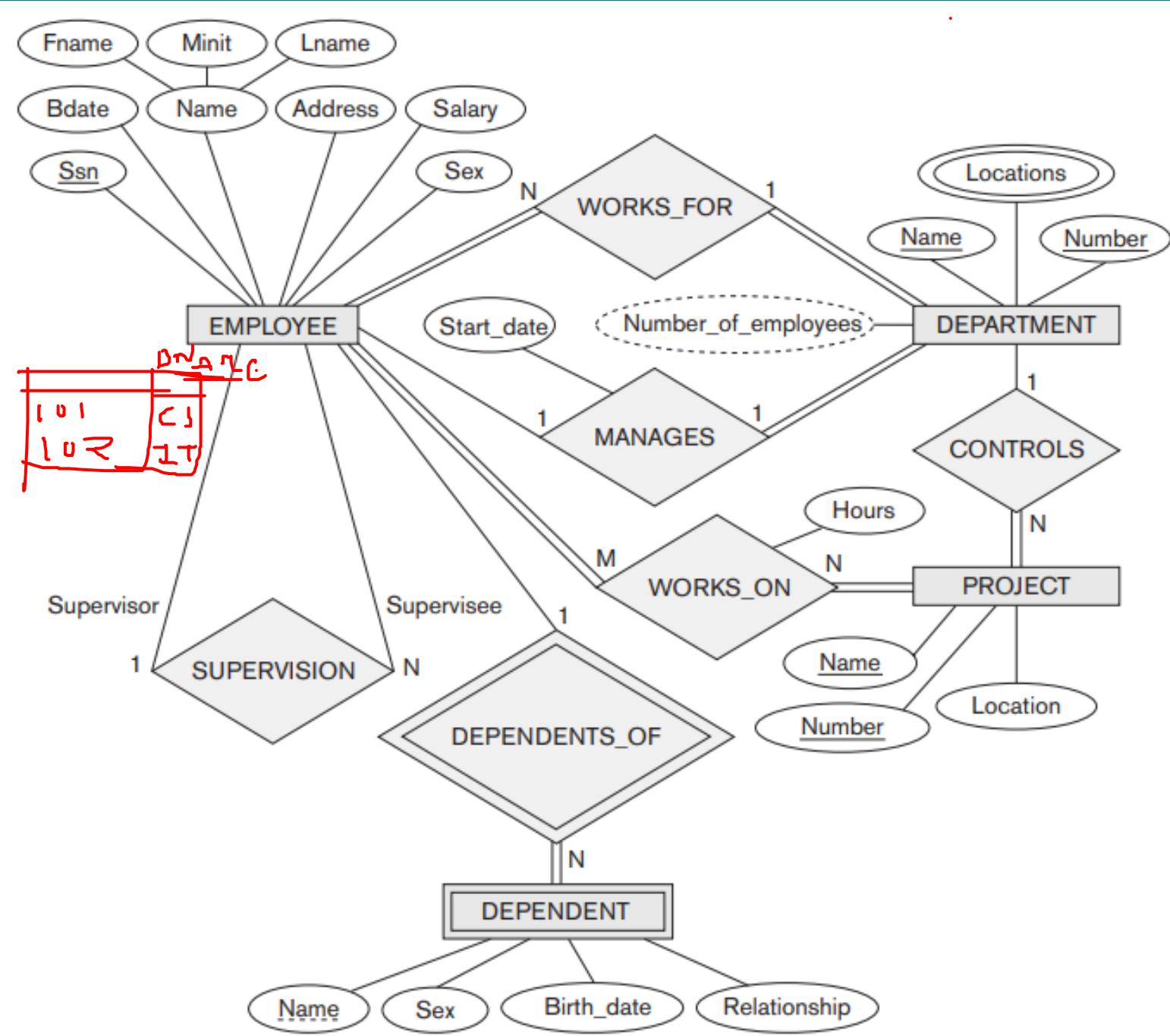


Total Participation of E_2 in R

Cardinality Ratio 1: N for $E_1 : E_2$ in R

Structural Constraint (min, max) on Participation of E in R

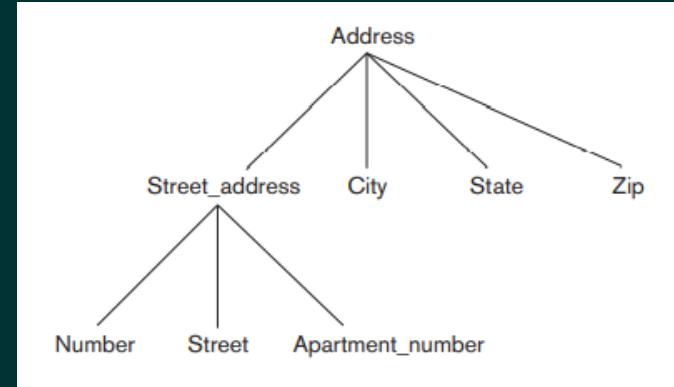
Data Model: ER Diagram



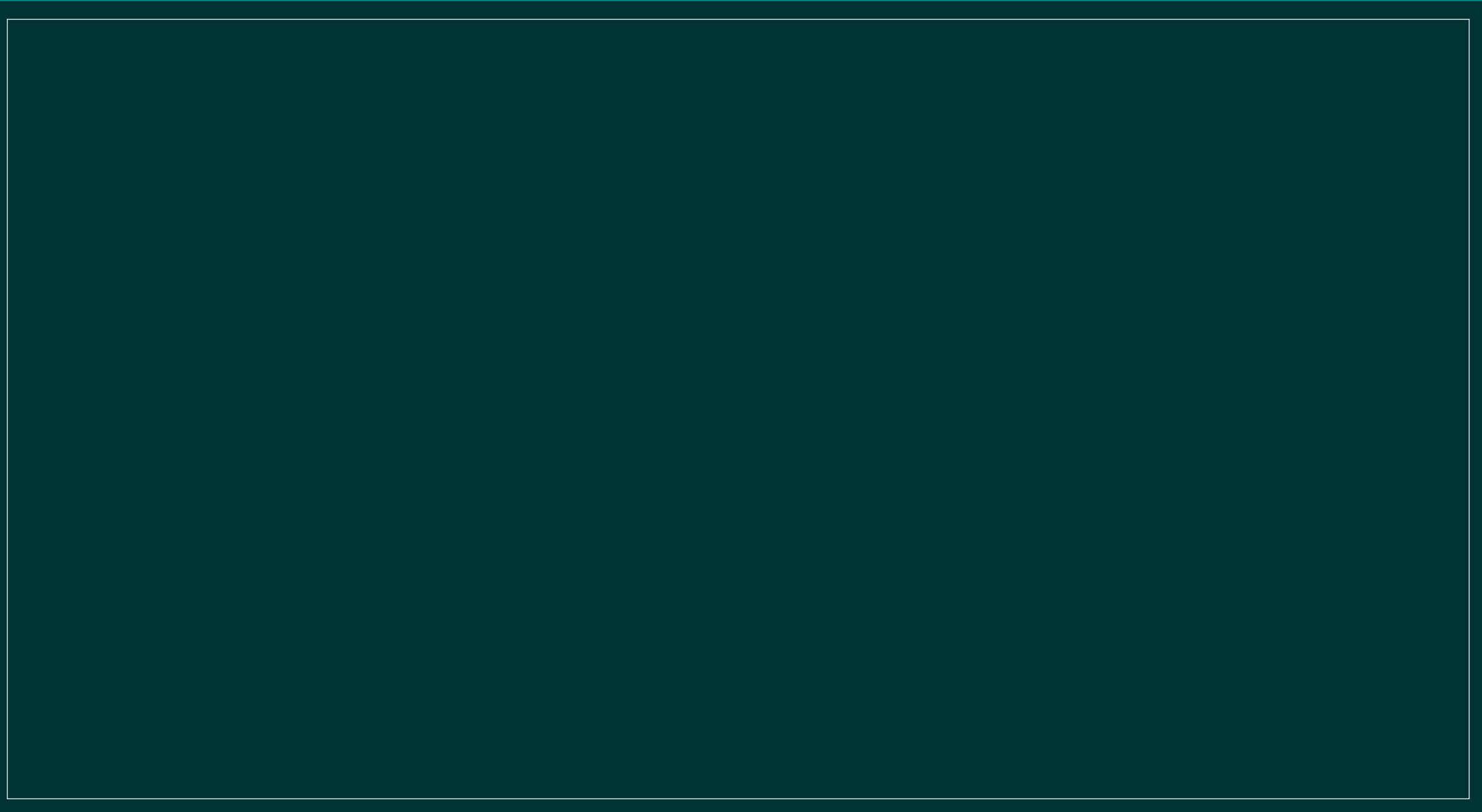
Data Model: Conceptual data model

Attribute Types

- Composite versus Simple (Atomic) Attributes
- Single-Valued versus Multivalued Attributes
- Stored versus Derived Attributes
- NULL Values
- Complex Attributes



Defunkt : [Address {—, phone }]



Relational Model

Table

STUDENT				
Name	Student_number	Class	Major	
R	92			
COURSE				
Course_name	Course_number	Credit_hours	Department	
PREREQUISITE				
Course_number	Prerequisite_number			
SECTION				
Section_identifier	Course_number	Semester	Year	Instructor
GRADE_REPORT				
Student_number	Section_identifier	Grade		

Relational Model

- Basic Unit of relational model is relation/table

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

Relation model

- Relations are unordered
- Database consists of multiple relations

<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

Relational Model

$$R = (A_1, A_2, A_3)$$

- Basic Unit of relational model is relation/table
- Each row is relation instance of a relation also called as tuple
- Each column represents attribute
 - Set of permitted atomic values of each attribute is called domain of attribute
 - Attribute values are (normally) required to be **atomic**; that is, indivisible
 - The special value **null** is a member of every domain
- **Relation Schema** : (Attribute set: Domain of the attribute) (Relation Intension)
 - $R = (A_1:D_1, A_2:D_2, \dots, A_n:D_n)$
 - Ex: *Instructor* = (*ID:int*, *name:String*, *dept_name:String*, *salary: float*)
- The degree (or arity) of a relation is the number of attributes n of its relation schema
- **Relation instance** corresponds to the values which the variable takes (Relation extension)
 - $r = \{t_1, t_2, \dots, t_m\}$
 - $t = \langle v_1, v_2, \dots, v_n \rangle$

Relational Model Notation

- A relation schema R of degree n is denoted by $R(A_1, A_2, \dots, A_n)$
- The uppercase letters Q, R, S denote relation names.
- The lowercase letters q, r, s denote relation states.
- The letters t, u, v denote tuples
- Relation schema such as STUDENT also indicates the current set of tuples in that relation—the current relation state—whereas STUDENT(Name, Ssn, ...) refers only to the relation schema.

R

Relational Model Constraints

- Constraints : restrictions on the actual values in a database state.
 - These constraints are derived from the rules in the miniworld that the database
- Constraints on databases can generally be divided into three main categories:
 - Constraints that are inherent in the data model (**Model-based constraints** or implicit constraints)
 - Constraints that can be directly expressed in the schemas of the data model, typically by specifying them in the DDL (**Schema-based constraints** or explicit constraints)
 - Constraints that cannot be directly expressed in the schemas of the data model, and hence must be expressed and enforced by the application programs or in some other way. (**Application-based** or semantic constraints or business rules)

The schema-based constraints include

- Domain constraints:
 - Domain constraints specify that within each tuple, the value of each attribute A must be an atomic value from the domain $\text{dom}(A)$
- Key constraints : No two tuples can have the same combination of values for all their attributes (Super key, candidate key and primary key)
- Constraints on NULLs: This constraint on attributes specifies whether NULL values are or are not permitted.
- Entity integrity constraints: entity integrity constraint states that no primary key value can be NULL
- Referential integrity constrain:
 - The referential integrity constraint is specified between two relations and is used to maintain the consistency among tuples in the two relations.
 - Informally, the referential integrity constraint states that a tuple one relation that refers to another relation must refer to an existing tuple in that relation

Key constraints

- A relation is defined as a set of tuples
- All elements of a set are distinct
- All tuples in a relation must also be distinct.
- Subsets of attributes of a relation schema R with the property that no two tuples in any relation state r of R should have the same combination of values for these attributes.
- Let subset of attributes be SK; then for any two distinct tuples t₁ and t₂ in a relation state r of R,
 - we have the constraint that: $t_1[\underline{SK}] \neq t_2[\underline{SK}]$
 - Any such set of attributes SK is called a super key of the relation schema R.
 - A super key SK specifies a uniqueness constraint that no two distinct tuples in any state r of R can have the same value for SK.
 - Every relation has at least one default super key – the set of all its attributes

Key

(Ssn, name, addr)

Candidate

- A key k of a relation schema R is a superkey of R with the properties
 - uniqueness property ✓
 - minimal superkey ✓
- A relation can have more than one key : Each one called as **candidate key**
- One of the candidate keys is identified as **Primary key**

(Emp_id, dept_id, proj_id)

1	2	3	1	1	1
1	2	2	1	1	1
2	1	3	2	2	3
2	2	1	2	2	1

Referential integrity constraint/Foreign Key

- A set of attributes FK in relation schema R1 is a foreign key of R1 that references relation R2 if it satisfies the following rules:
 - The attributes in FK have the same domain(s) as the primary key attributes PK of R2; the attributes FK are said to reference or refer to the relation R2.
 - A value of FK in a tuple t1 of the current state $r_1(R1)$ either occurs as a value of PK for some tuple t2 in the current state $r_2(R2)$ or is NULL. In the former case, we have $t1[FK] = t2[PK]$, and we say that the tuple t1 references or refers to the tuple t2.
- R1 is called the referencing relation and R2 is the referenced relation.
- If these two conditions hold, a referential integrity constraint from R1 to R2 is said to hold.
- In a database of many relations, there are usually many referential integrity constraints.

EMPLOYEE

Fname	Minit	Lname	<u>Ssn</u>	Bdate	Address	Sex	Salary	Super_ssn	Dno
-------	-------	-------	------------	-------	---------	-----	--------	-----------	-----

DEPARTMENT

Dname	<u>Dnumber</u>	Mgr_ssn	Mgr_start_date
-------	----------------	---------	----------------

DEPT_LOCATIONS

<u>Dnumber</u>	<u>Dlocation</u>
----------------	------------------

PROJECT

Pname	<u>Pnumber</u>	Plocation	<u>Dnum</u>
-------	----------------	-----------	-------------

WORKS_ON

<u>Essn</u>	<u>Pno</u>	Hours
-------------	------------	-------

DEPENDENT

<u>Essn</u>	Dependent_name	Sex	Bdate	Relationship
-------------	----------------	-----	-------	--------------

EMPLOYEE

Fname	Minit	Lname	<u>Ssn</u>	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	<u>Dnumber</u>	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

<u>Dnumber</u>	<u>Dlocation</u>
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

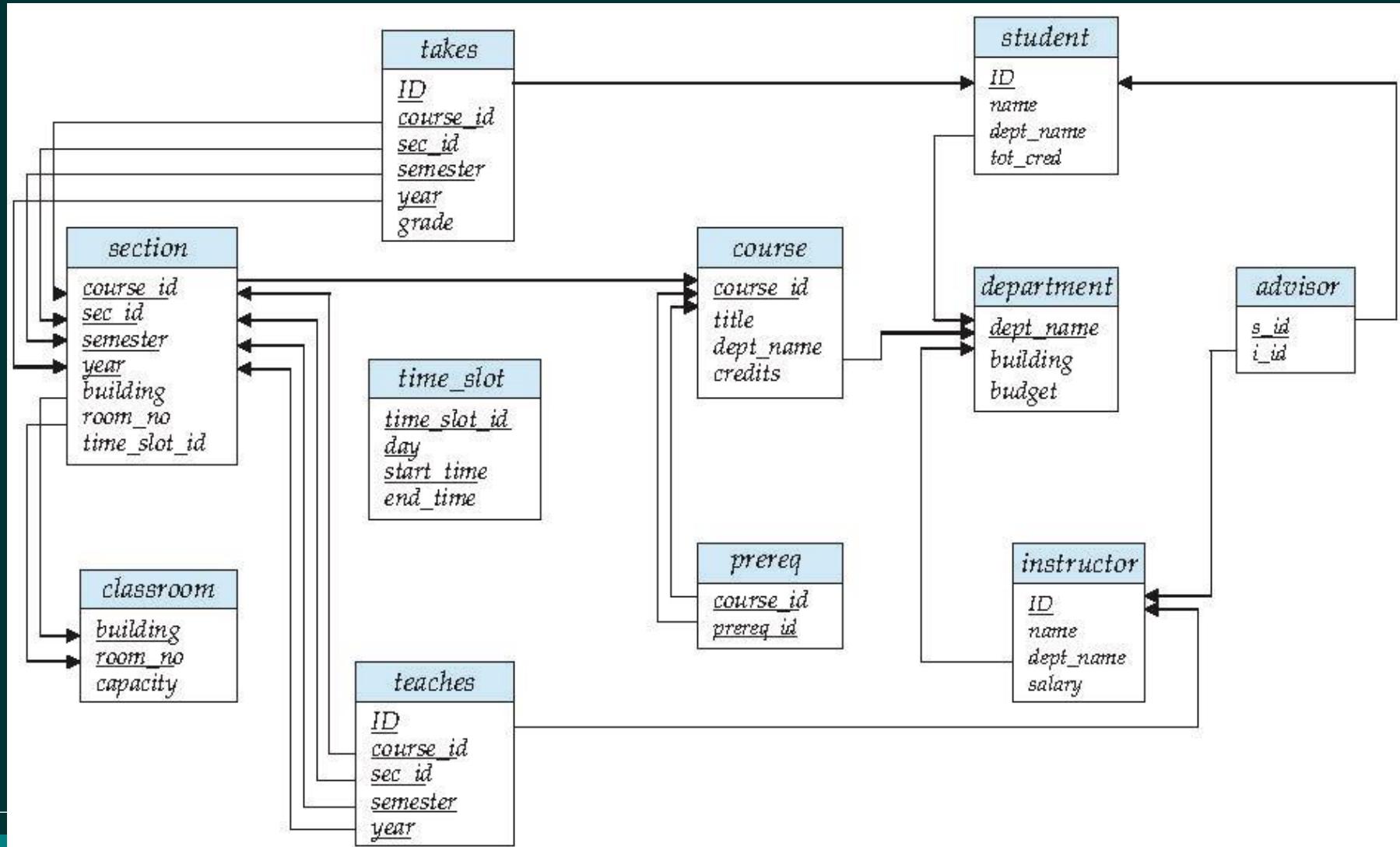
<u>Essn</u>	<u>Pno</u>	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0

PROJECT

Pname	<u>Pnumber</u>	Plocation	<u>Dnum</u>
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

Example University schema diagram



Exercise

Consider the following relations for a database that keeps track of student enrollment in courses and the books adopted for each course: Identify the primary and foreign keys

- STUDENT(Ssn, Name, Major, Bdate)
- COURSE(Course#, Cname, Dept)
- ENROLL(Ssn, Course#, Quarter, Grade)
- BOOK_ADOPTION(Course#, Quarter, Book_isbn)
- TEXT(Book isbn, Book_title, Publisher, Author)

Relational Query Languages

- A query language is a language in which a user requests information from the database.
- Procedural: The user instructs the system to perform a sequence of operations on the database to compute the desired result.
- The result of a query is a new relation that is a subset of the original relation
 - Example, if we select tuples from the instructor relation satisfying the predicate “salary is greater than \$85000”

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
12121	Wu	Finance	90000
22222	Einstein	Physics	95000
33456	Gold	Physics	87000
83821	Brandt	Comp. Sci.	92000

Database Languages

- **Data Definition Language**

- Specification notation for defining the database schema

- `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)

- Referential integrity (**references** constraint in SQL)

- e.g. `dept_name` value in any *instructor* tuple must appear in *department* relation

- Authorization

Database Languages

Data Manipulation Language

- Language for accessing and manipulating the data organized by the appropriate data model
 - DML also known as query language
- Two classes of languages
 - **Procedural** – user specifies what data is required and how to get those data
 - **Declarative (nonprocedural)** – user specifies what data is required without specifying how to get those data
- SQL is the most widely used query language

Database Languages

- **SQL**: widely used non-procedural language
 - Example: Find the name of the instructor with ID 22222
select *name* **from** *instructor* **where** *instructor.ID* = '22222'
 - Example: Find the ID and building of instructors in the Physics dept.
select *instructor.ID, department.building*
from *instructor, department*
where *instructor.dept_name* = *department.dept_name* **and**
 department.dept_name = 'Physics'
- 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 Languages

The Relational Algebra

- The relational algebra is a procedural query language.
- It consists of a set of operations that take one or two relations as input and produce a new relation as their result.

Relational Operations

- Selection : Selection of rows based on some condition
- Projection : Selection of columns
- Joining : Joining two or more relations
- Union
- Difference
- Intersection
- Natural Join

Relational Operations

- Selection : σ

Selection of rows based on some condition

$$\sigma_{\underline{A=B \text{ and } D > 5}} (r)$$

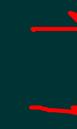
□ Relation r

$$\cancel{\sigma} =$$

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

Select tuples with $A=B$ and $D > 5$

$$\sigma_{\underline{A=B \text{ and } D > 5}} (r)$$



A	B	C	D
α	α	1	7
β	β	23	10

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

Figure 3.6 The *loan* relation.

◦ $\sigma_{\text{branch-name} = \text{"Perryridge"} \wedge \text{amount} > 1200}(\text{loan})$

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

Figure 3.6 The *loan* relation.

$$\sigma_{\text{branch-name} = \text{"Perryridge"} \wedge \text{amount} > 1200}(\text{loan})$$

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
L-15	Perryridge	1500
L-16	Perryridge	1300

Relational Operations

- Projection : Π
- Selection of columns

$$\Pi_{A, C}(r)$$

- Relation r :

A	B	C
α	10	1
α	20	1
β	30	1
β	40	2

A	C
α	1
α	1
β	1
β	2

=

A	C
α	1
β	1

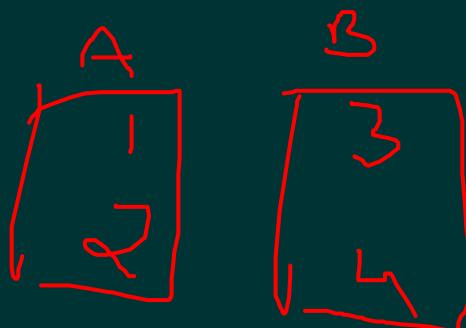
<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
<u>Hayes</u>	Main	Harrison
Johnson	Alma	Palo Alto
<u>Jones</u>	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton

Figure 3.4 The *customer* relation.

$$\Pi_{\text{customer-name}} (\sigma_{\text{customer-city} = \text{"Harrison"}} (\underline{\text{customer}}))$$


Relational Operations

- Joining : Joining two relations - Cartesian Product



A	B
1	3
2	3
1	4
2	4

□ Relations r, s :

A	B
α	1
β	2

r

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

□ $r \times s$:

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b



$r \times s$

Relational Operations

- Union
- Relations r, s :

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

□ $r \cup s$:

A	B
α	1
α	2
β	1
β	3

<i>customer-name</i>	<i>loan-number</i>
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

Figure 3.7 The *borrower* relation.

◦ $\Pi_{\text{customer-name}} (\text{borrower}) \cup \Pi_{\text{customer-name}}^c (\text{depositor})$

<i>customer-name</i>	<i>account-number</i>
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305

Figure 3.5 The *depositor* relation.

<i>customer-name</i>	<i>loan-number</i>
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

Figure 3.7 The *borrower* relation.

<i>customer-name</i>	<i>account-number</i>
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305

Figure 3.5 The *depositor* relation.

$$\circ \Pi_{\text{customer-name}} (\text{borrower}) \cup \Pi_{\text{customer-name}} (\text{depositor})$$

<i>customer-name</i>
Adams
Curry
Hayes
Jackson
Jones
Smith
Williams
Lindsay
Johnson
Turner

Relational Operations

- Difference

- Relations r, s :

□ $r - s$:

$r - s$ $s - r$

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

A	B
α	1
β	1

✓

Relational Operations

- Intersection : \cap

- Relation r, s :

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- $r \cap s$

A	B
α	2

<i>customer-name</i>	<i>loan-number</i>
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

Figure 3.7 The *borrower* relation.

<i>customer-name</i>	<i>account-number</i>
✓ Hayes	A-102
Johnson	A-101
Johnson	A-201
✓ Jones	A-217
Lindsay	A-222
✓ Smith	A-215
Turner	A-305

Figure 3.5 The *depositor* relation.

- $\Pi_{\text{customer-name}} (\text{borrower}) \cap \Pi_{\text{customer-name}}^c (\text{depositor})$

The Set Difference Operation

- denoted by –
- allows us to find tuples that are in one relation but are not in another.
- The expression $r - s$ produces a relation containing those tuples in r but not in s .

<i>customer-name</i>	<i>loan-number</i>
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

Figure 3.7 The *borrower* relation.

<i>customer-name</i>	<i>account-number</i>
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305

Figure 3.5 The *depositor* relation.

- cl tip
 $\circ \Pi_{\text{customer-name}} (\text{borrower}) - \Pi_{\text{customer-name}} (\text{depositor})$ Borrowing

<i>customer-name</i>
Johnson
Lindsay
Turner

The Cartesian-Product Operation

- The Cartesian-product operation, denoted by a cross (\times)
- Allows us to combine information from any two relations.
- We write the Cartesian product of relations r_1 and r_2 as $r_1 \times r_2$.
- $r = \underline{\text{borrower}} \times \underline{\text{loan}}$ is $(\text{borrower}.\text{customer-name}, \text{borrower}.\text{loan-number}, \text{loan}.\text{loan-number}, \text{loan}.\text{branch-name}, \text{loan}.\text{amount})$

<i>customer-name</i>	<i>loan-number</i>
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

Figure 3.7 The *borrower* relation.

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

Figure 3.6 The *loan* relation.

The Cartesian-Product

- $r = \text{borrower} \times \text{loan}$:
 $(\text{borrower}.\text{customer-name}, \text{borrower}.\text{loan-number}, \text{loan}.\text{loan-number}, \text{loan}.\text{branch-name}, \text{loan}.\text{amount})$

<i>customer-name</i>	<i>loan-number</i>
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

Figure 3.7 The *borrower* relation.

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

Figure 3.6 The *loan* relation.

<i>customer-name</i>	<i>borrower.</i> <i>loan-number</i>	<i>loan.</i> <i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
Adams	L-16	L-11	Round Hill	900
Adams	L-16	L-14	Downtown	1500
Adams	L-16	L-15	Perryridge	1500
Adams	L-16	L-16	Perryridge	1300
Adams	L-16	L-17	Downtown	1000
Adams	L-16	L-23	Redwood	2000
Adams	L-16	L-93	Mianus	500
Curry	L-93	L-11	Round Hill	900
Curry	L-93	L-14	Downtown	1500
Curry	L-93	L-15	Perryridge	1500
Curry	L-93	L-16	Perryridge	1300
Curry	L-93	L-17	Downtown	1000
Curry	L-93	L-23	Redwood	2000
Curry	L-93	L-93	Mianus	500
Hayes	L-15	L-11	900	
Hayes	L-15	L-14	1500	
Hayes	L-15	L-15	1500	
Hayes	L-15	L-16	1300	
Hayes	L-15	L-17	1000	
Hayes	L-15	L-23	2000	
Hayes	L-15	L-93	500	
...
...
...
Smith	L-23	L-11	Round Hill	900
Smith	L-23	L-14	Downtown	1500
Smith	L-23	L-15	Perryridge	1500
Smith	L-23	L-16	Perryridge	1300
Smith	L-23	L-17	Downtown	1000
Smith	L-23	L-23	Redwood	2000
Smith	L-23	L-93	Mianus	500
Williams	L-17	L-11	Round Hill	900
Williams	L-17	L-14	Downtown	1500
Williams	L-17	L-15	Perryridge	1500
Williams	L-17	L-16	Perryridge	1300
Williams	L-17	L-17	Downtown	1000
Williams	L-17	L-23	Redwood	2000
Williams	L-17	L-93	Mianus	500

Figure 3.14 Result of *borrower* \times *loan*.

The Cartesian-Product

- $r = \sigma_{\text{branch-name} = \text{"Perryridge"}}(\text{borrower} \times \text{loan})$

<i>customer-name</i>	<i>loan-number</i>
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

Figure 3.7 The *borrower* relation.

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

Figure 3.6 The *loan* relation.

<i>customer-name</i>	<i>borrower.</i> <i>loan-number</i>	<i>loan.</i> <i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
Adams	L-16	L-11	Round Hill	900
Adams	L-16	L-14	Downtown	1500
Adams	L-16	L-15	Perryridge	1500
Adams	L-16	L-16	Perryridge	1300
Adams	L-16	L-17	Downtown	1000
Adams	L-16	L-23	Redwood	2000
Adams	L-16	L-93	Mianus	500
Curry	L-93	L-11	Round Hill	900
Curry	L-93	L-14	Downtown	1500
Curry	L-93	L-15	Perryridge	1500
Curry	L-93	L-16	Perryridge	1300
Curry	L-93	L-17	Downtown	1000
Curry	L-93	L-23	Redwood	2000
Curry	L-93	L-93	Mianus	500
Hayes	L-15	L-11	Round Hill	900
Hayes	L-15	L-14	Downtown	1500
Hayes	L-15	L-15	Perryridge	1500
Hayes	L-15	L-16	Perryridge	1300
Hayes	L-15	L-17	Downtown	1000
Hayes	L-15	L-23	Redwood	2000
Hayes	L-15	L-93	Mianus	500
...
...
...
Smith	L-23	L-11	Round Hill	900
Smith	L-23	L-14	Downtown	1500
Smith	L-23	L-15	Perryridge	1500
Smith	L-23	L-16	Perryridge	1300
Smith	L-23	L-17	Downtown	1000
Smith	L-23	L-23	Redwood	2000
Smith	L-23	L-93	Mianus	500
Williams	L-17	L-11	Round Hill	900
Williams	L-17	L-14	Downtown	1500
Williams	L-17	L-15	Perryridge	1500
Williams	L-17	L-16	Perryridge	1300
Williams	L-17	L-17	Downtown	1000
Williams	L-17	L-23	Redwood	2000
Williams	L-17	L-93	Mianus	500

Figure 3.14 Result of *borrower* × *loan*.

The Cartesian-Product

- $r = \sigma_{\text{branch-name} = \text{"Perryridge"}}(\text{borrower} \times \text{loan})$

<i>customer-name</i>	<i>loan-number</i>
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

Figure 3.7 The *borrower* relation.

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

Figure 3.6 The *loan* relation.

<i>customer-name</i>	<i>borrower.loan-number</i>	<i>loan.loan-number</i>	<i>branch-name</i>	<i>amount</i>
Adams	L-16	L-15	Perryridge	1500
Adams	L-16	L-16	Perryridge	1300
Curry	L-93	L-15	Perryridge	1500
Curry	L-93	L-16	Perryridge	1300
Hayes	L-15	L-15	Perryridge	1500
Hayes	L-15	L-16	Perryridge	1300
Jackson	L-14	L-15	Perryridge	1500
Jackson	L-14	L-16	Perryridge	1300
Jones	L-17	L-15	Perryridge	1500
Jones	L-17	L-16	Perryridge	1300
Smith	L-11	L-15	Perryridge	1500
Smith	L-11	L-16	Perryridge	1300
Smith	L-23	L-15	Perryridge	1500
Smith	L-23	L-16	Perryridge	1300
Williams	L-17	L-15	Perryridge	1500
Williams	L-17	L-16	Perryridge	1300

Figure 3.15 Result of $\sigma_{\text{branch-name} = \text{"Perryridge"}}(\text{borrower} \times \text{loan})$.

Rename (alias) operator ($\underline{\rho}$)

- Find the largest account balance in the bank.



account-number	branch-name	balance
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900
A-215	Mianus	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Round Hill	350

Figure 3.1 The *account* relation.

- $(\prod_{\text{account.balance}} (\sigma_{\text{account.balance} < d.\text{balance}} (\text{account} \times \underline{\rho}_d (\text{account}))))$
- $\prod_{\text{balance}} (\text{account}) - [\prod_{\text{account.balance}} (\sigma_{\text{account.balance} < d.\text{balance}} (\text{account} \times \underline{\rho}_d (\text{account})))]$

Rename

Exercise

- Customers who live on the same street and in the same city as Smith.
- $\Pi_{\text{customer}}.\text{customer-name}$
- $(\sigma_{\text{customer.customer-street} = \text{smith-addr.street}} \wedge \sigma_{\text{customer.customer-city} = \text{smith-addr.city}} (\text{customer} \times \rho_{\text{smith-addr(street,city)}}))$
- $(\Pi_{\text{customer-street, customer-city}} (\sigma_{\text{customer-name} = "Smith"}(\text{customer}))))$

<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
Hayes	Main	Harrison
Johnson	Alma	Palo Alto
Jones	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton

Figure 3.4 The *customer* relation.

◦ $\Pi_{\text{customer}}.\text{customer-name} (\sigma_{\text{customer.customer-street}=\text{smith-addr.street}} \wedge \sigma_{\text{customer.customer-city}=\text{smith-addr.city}} (\text{customer} \times \rho_{\text{smith-addr(street,city)}} (\Pi_{\text{customer-street, customer-city}} (\sigma_{\text{customer-name} = "Smith"}(\text{customer})))))$

1
—

Symbol (Name)	Example of Use
σ (Selection)	$\sigma \text{ salary} >= 85000 (\text{instructor})$ Return rows of the input relation that satisfy the predicate.
Π (Projection)	$\Pi_{ID, \text{salary}} (\text{instructor})$ Output specified attributes from all rows of the input relation. Remove duplicate tuples from the output.
\bowtie (Natural Join)	$\text{instructor} \bowtie \text{department}$ Output pairs of rows from the two input relations that have the same value on all attributes that have the same name.
\times (Cartesian Product)	$\text{instructor} \times \text{department}$ Output all pairs of rows from the two input relations (regardless of whether or not they have the same values on common attributes)
\cup (Union)	$\Pi_{name}(\text{instructor}) \cup \Pi_{name}(\text{student})$ Output the union of tuples from the two input relations.

Relational Operations

- Natural Join: \bowtie
- The natural join operation on two relations matches tuples whose values are the same on all attribute names that are common to both relations.

◦ Relations r, s:

- Natural Join
 - $r \bowtie s$

A	B	C	D	
α	1	α	a	
β	2	γ	a	
γ	4	β	b	
α	1	γ	a	
δ	2	β	b	

r

B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	ε

s

A	B	C	D	E
α	1	α	a	α
α	1	α	a	γ
α	1	γ	a	α
α	1	γ	a	γ
δ	2	β	b	δ

Natural Join :

- “Find the names of all customers who have a loan at the bank, along with the loan number and the loan amount.” $\sigma_{\text{borrower.loan-number} = \text{loan.loan-number}}$
- $\Pi_{\text{customer-name}, \text{loan.loan-number}, \text{amount}} (\sigma_{\text{borrower.loan-number} = \text{loan.loan-number}} (\text{borrower} \times \text{loan}))$

customer-name	loan-number
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

Figure 3.7 The *borrower* relation.

loan-number	branch-name	amount
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

Figure 3.6 The *loan* relation.

customer-name	loan-number	amount
Adams	L-16	1300
Curry	L-93	500
Hayes	L-15	1500
Jackson	L-14	1500
Jones	L-17	1000
Smith	L-23	2000
Smith	L-11	900
Williams	L-17	1000

Figure 3.21 Result of $\Pi_{\text{customer-name}, \text{loan-number}, \text{amount}} (\text{borrower} \bowtie \text{loan})$.

- Let the following relation schemas be given: $R = (A, B, C)$ and $S = (D, E, F)$
- Let relations $\underline{r(R)}$ and $\underline{s(S)}$ be given.
- Interpret each of the following:

a. $\underline{\Pi_A(r)} \Rightarrow$

b. $\underline{\sigma_{B=17}(r)} \checkmark =$

c. $\underline{r \times s} \quad \checkmark$

d. $\underline{\Pi_{A,F}(\sigma_{C=D}(r \times s))}$

$\sigma_{\text{salary} > 12000} (\text{employee} \times \text{work})$

Exercise

Database

- employee (person-name, street, city)
- works (person-name, company-name, salary)
- company (company-name, city)
- manages (person-name, manager-name)

Write relational algebra notation for

- Fetching all the employee details having salary greater than 20000

=====

$$\sigma_{\text{salary} > 12000}(\text{employee} \bowtie \text{work})$$

The Assignment Operation

- The assignment operation, denoted by \leftarrow , works like assignment in a programming language.
- $\text{temp1} \leftarrow \Pi_{R-S}(r)$
- $\text{temp2} \leftarrow \Pi_{R-S}((\text{temp1} \times s) - \Pi_{R-S,S}(r))$
- $\text{result} = \text{temp1} - \text{temp}$



QUIZ



ICT 3157: DATABASE SYSTEMS

3. Introduction to SQL
Girija Attigeri

Contents

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

History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
- SQL-86, SQL-89, SQL-92
- SQL:1999, SQL:2003, SQL:2008
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.

Data Definition Language

The SQL **data-definition language (DDL)** allows the specification of information about relations, including:

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints
- And other information such as
 - The set of indices to be maintained for each relations.
 - Security and authorization information for each relation.
 - The physical storage structure of each relation on disk.

Domain Types in SQL

- **char(n)** : Fixed length character string, with user-specified length n.
- **varchar(n)**: Variable length character strings, with user-specified maximum length n.
- **Int**: Integer (a finite subset of the integers that is machine-dependent).
- **Smallint**: Small integer (a machine-dependent subset of the integer domain type).
- **numeric(p,d)**: Fixed point number, with user-specified precision of 'p' digits, with 'd' digits to the right of decimal point.
- **real, double precision**: Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(n)**: Floating point number, with user-specified precision of at least n digits.

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

Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a large object
- **blob**: binary large object -- object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
- **clob**: character large object -- object is a large collection of character data
- When a query returns a large object, a pointer is returned rather than the large object itself.

Create Table Construct

- Create database university;
- Use university
- An SQL relation is defined using the create table command:
- `create table r (A1 D1, A2 D2, ..., An Dn, (integrity-constraint1),..., (integrity-constraintk))`
- `r` is the name of the relation
- each `Ai` is an attribute name in the schema of relation `r`
- `Di` is the data type of values in the domain of attribute `Ai`

Example:

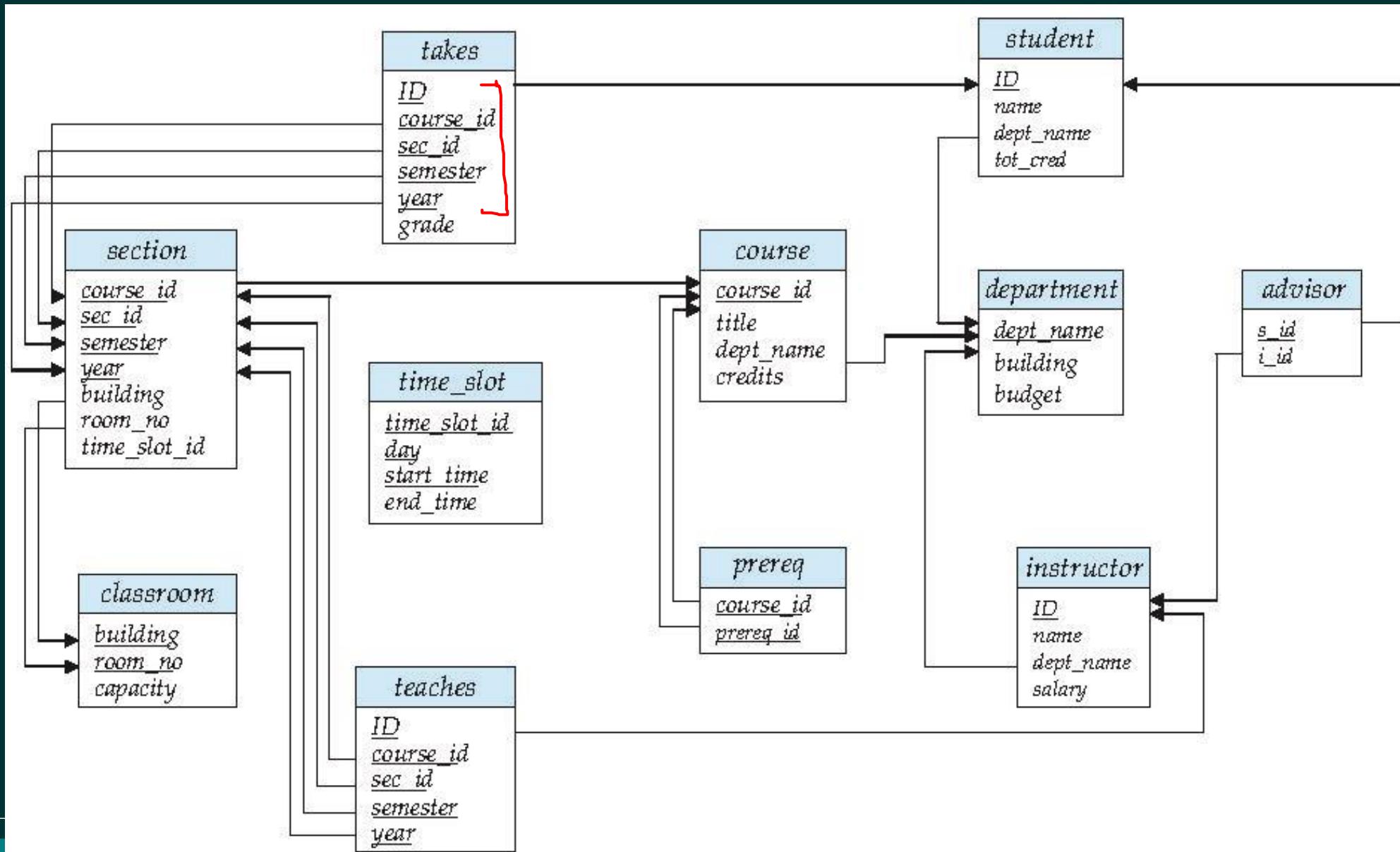
- `create table instructor (ID char(5), name varchar(20) not null, dept_name varchar(20), salary numeric(8,2))`
- `insert into instructor values ('10211', 'Smith', 'Biology', 66000);`

Integrity Constraints in Create Table

- not null ✓
- primary key (A₁, ..., A_n) ✓
- foreign key (A_m, ..., A_n) references r
- Example: Declare ID as the primary key for instructor.
- ```
create table instructor (
 ID char(5),
 name varchar(20) not null,
 dept_name varchar(20),
 salary numeric(8,2),
 primary key (ID), foreign key (dept_name) references department)
```


- **primary key** declaration on an attribute automatically ensures **not null**

# Schema Diagram for University Database



- create table student (  
    ID                varchar(5),  
    name              varchar(20) not null,  
    dept\_name        varchar(20),  
    tot\_cred         numeric(3,0),  
    primary key (ID),  
    foreign key (dept\_name) references department );
- create table takes (  
    ID                varchar(5),  
    course\_id        varchar(8),  
    sec\_id            varchar(8),  
    semester         varchar(6),  
    year              numeric(4,0),  
    grade             varchar(2),  
    primary key (ID, course\_id, sec\_id, semester, year),  
    foreign key (ID) references student,  
    foreign key (course\_id, sec\_id, semester, year) references section );
- Note: sec\_id can be dropped from primary key above, to ensure a student cannot be registered for two sections of the same course in the same semester

- ```
create table course (
    course_id      varchar(8) primary key,
    title          varchar(50),
    dept_name      varchar(20),
    credits         numeric(2,0),
    foreign key (dept_name) references department);
```
- Primary key declaration can be combined with attribute declaration as shown above

Drop and Alter Table Constructs

- **drop table** student
 - Deletes the table and its contents
- **delete from** student
 - Deletes all contents of table, but retains table
- **alter table**
 - alter table r add A D
 - where A is the name of the attribute to be added to relation r and D is the domain of A.
 - All tuples in the relation are assigned null as the value for the new attribute.
 - alter table r drop A
 - where A is the name of an attribute of relation r
 - Dropping of attributes not supported by many databases

Drop and Alter Table Constructs

- alter table
 - `Alter table table_Name Modify Col_Name datatype constraint(s)`
 - `Alter table r modify A D C(s)`
 - where A is the name of the attribute to be added to relation r and D is the domain of A.
 - `Desc TableName;`
 - `Select Table_Name from User_Tables;`

Basic Query Structure

- The SQL data-manipulation language (DML) provides the ability to query information, and insert, delete and update tuples
- A typical SQL query has the form:

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

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

The select Clause

- The select clause list 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 names of all departments with instructor, and remove duplicates
 - **select distinct dept_name from instructor**
- The keyword all specifies that duplicates not be removed.
 - **select all dept_name from instructor**

The select Clause (Cont.)

- An asterisk in the select clause denotes “all attributes”
 - ```
select * from instructor
```
- 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.

# 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 with salary > 80000
  - ```
select name
      from instructor
      where dept_name = 'Comp. Sci.' and salary > 80000
```
- Comparison results can be combined using the logical connectives and, or, not
- Comparisons can be applied to results of arithmetic expressions.

The from Clause

- The from clause lists the relations involved in the query
- Find the Cartesian product instructor X teaches
 - select * from **instructor, teaches**
 - **generates every possible instructor – teaches pair**, with all attributes from both relations
- Cartesian product not very useful directly, but useful combined with where-clause condition (selection operation in relational algebra)

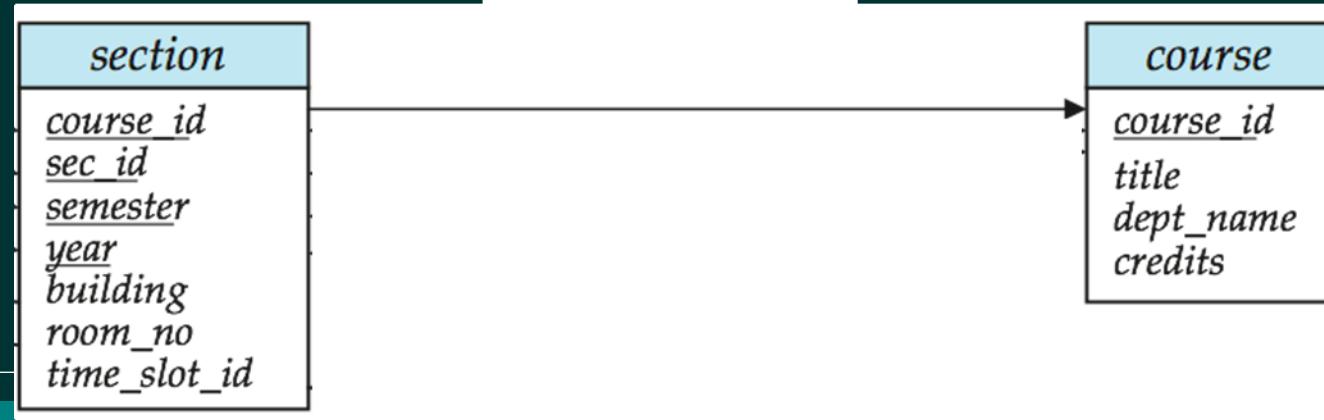
Cartesian Product: instructor X teaches

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
22456	Gill	Physics	87000

<i>ID</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009

Joins

- For all instructors who have taught some course, find their names and the course ID of the courses they taught.
 - ```
select name, course_id
 from instructor, teaches
 where instructor.ID = teaches.ID
```
- Find the course ID, semester, year and title of each course offered by the Comp. Sci. department
  - ```
select section.course_id, semester, year, title
      from section, course
     where section.course_id = course.course_id and dept_name = 'Comp. Sci.'
```



Natural Join

- Natural join matches tuples with the same values for all common attributes, and retains only one copy of each common column
 - select * from instructor natural join teaches;

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010

Natural Join Example

- List the names of instructors along with the course ID of the courses that they taught.
- In two ways we can write the Query:

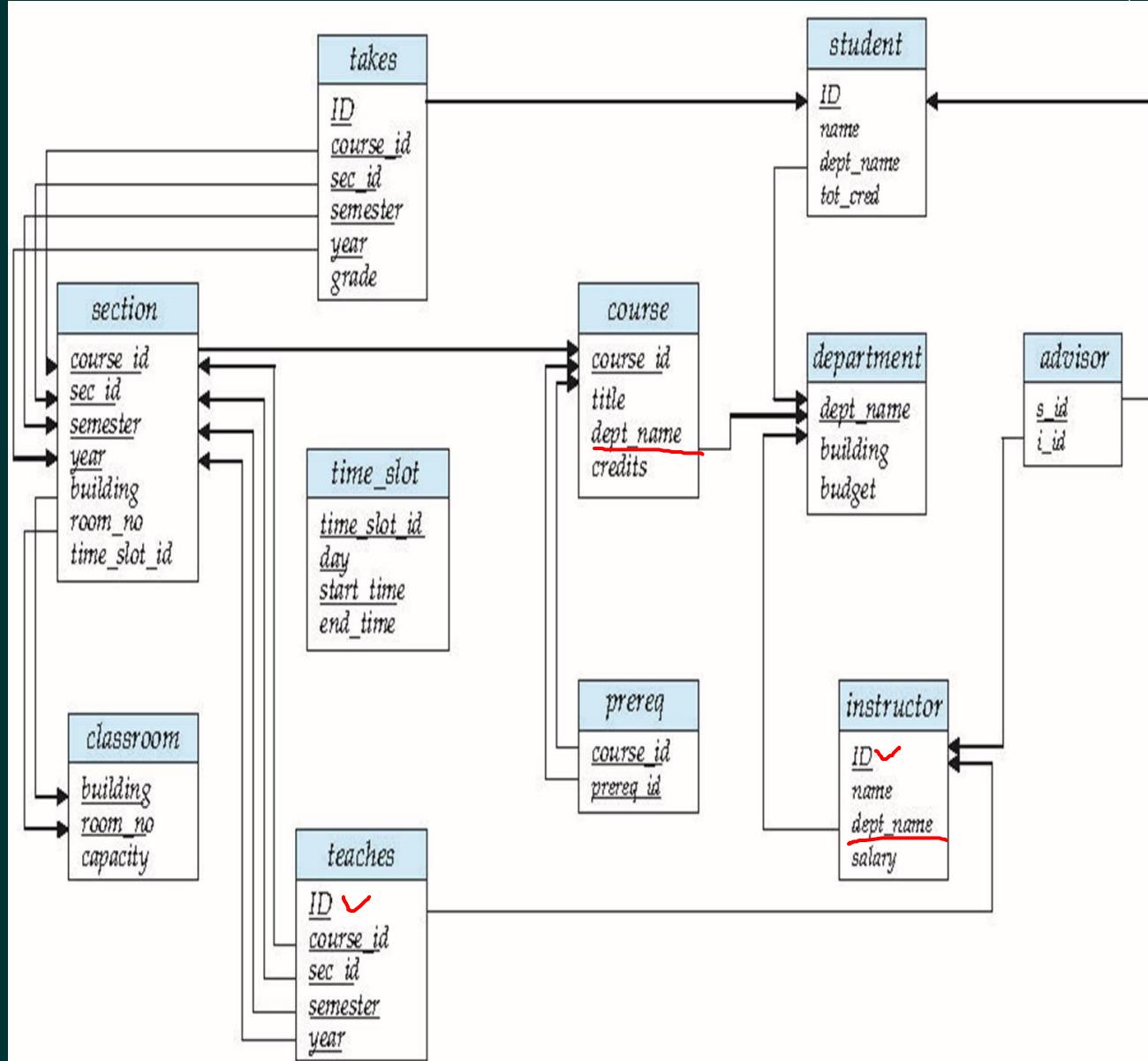
```
select name, course_id  
from instructor, teaches  
where instructor.ID = teaches.ID;
```

OR

```
select name, course_id  
from instructor natural join teaches;
```

Natural Join (Cont.)

- Danger in natural join: beware of unrelated attributes with same name which get equated incorrectly
- List the names of instructors along with the titles of courses that they teach
 - Incorrect version (makes `course.dept_name = instructor.dept_name`)
 - [select name, title
from instructor natural join teaches
natural join course;]



Natural Join (Cont.)

- Danger in natural join: beware of unrelated attributes with same name which get equated incorrectly
- List the names of instructors along with the titles of courses that they teach
 - Incorrect version (makes course.dept_name = instructor.dept_name)
 - select name, title
from instructor natural join teaches natural join course;
- Correct version
 - select name, title
from ~~instructor~~ natural join ~~teaches~~, course
where teaches.course_id = course.course_id;
- Another correct version
 - select name, title
from (instructor natural join teaches)
join course using(course_id);



```
select name, title  
from instructor natural join teaches, course  
where teaches.course_id = course.course_id;
```



```
from (instructor natural join teaches)  
join course using(course_id);
```

The Rename Operation

- The SQL allows renaming relations and attributes using the as clause:
 - old-name as new-name
- E.g.

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

The Rename Operation

- 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
  - $\text{instructor as } T \equiv \text{instructor } T$
- Keyword as must be omitted in Oracle

# 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%'
- 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.

%. USE  
\_ ASK  
SE  
S\_E

|  
— — — — .|

# String Operations (Cont.)

- Match the string “100 %” // Usage of % in a string  
(eg. Retrieve whose attendance status is 100%)  
like '100 \%'  
escape '\'

- ‘\’ is used to specify the special character such as % or \.

- 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.

- Eg.: Select SUBSTR('ABCDEF', 2, 3) as substring from Dual ; o/p: BCD
  - select length(name) as length\_val from dual ;
  - select upper('hi') as Upper\_val from dual ;

# 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

- General Form:

- SELECT distinct name  
FROM instructor WHERE Dept\_name='CS' ORDER BY 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, \$90,000 and \$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');
```
 - All SQL platforms may not support this syntax

Set Operations

- Find courses that ran in Fall 2009 or in Spring 2010
 - (select course_id from section where sem = 'Fall' and year = 2009)
union
(select course_id from section where sem = 'Spring' and year = 2010)
- Find courses that ran in Fall 2009 and in Spring 2010
 - (select course_id from section where sem = 'Fall' and year = 2009)
intersect
(select course_id from section where sem = 'Spring' and year = 2010)
- Find courses that ran in Fall 2009 but not in Spring 2010
 - (select course_id from section where sem = 'Fall' and year = 2009)
except (Minus for Oracle)
(select course_id from section where sem = 'Spring' and year = 2010)

Set Operations

- Set operations union, intersect, and except
 - Each of the above operations automatically eliminates duplicates
 - To retain all duplicates use the corresponding multiset versions union all, intersect all and 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
```

# Null Values and Three Valued Logic

- Any comparison with null returns unknown
  - Example:  $5 < \text{null}$  or  $\text{null} <> \text{null}$  or  $\text{null} = \text{null}$
- Three-valued logic using the truth value unknown:
  - OR: (**unknown** or **true**) = **true**,  
          (**unknown** or **false**) = **unknown**  
          (**unknown** or **unknown**) = **unknown**
  - AND: (**true** and **unknown**) = **unknown**,  
          (**false** and **unknown**) = **false**,  
          (**unknown** and **unknown**) = **unknown**
  - NOT: (**not unknown**) = **unknown**
  - “P is **unknown**” evaluates to **true** if predicate P evaluates to **unknown**
- Result of **where** clause predicate is treated **as false if it evaluates to unknown**
- Eg. Where salary>50,000 and /or dept\_name='CS' ;

# Aggregate Functions

- These **functions operate on** the multi set 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
- To get the statistics of the relation we can use aggregate function.



# QUIZ



# ICT 3157: DATABASE SYSTEMS

4. Intermediate SQL  
Girija Attigeri

# Basic SQL

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

# Intermediate SQL

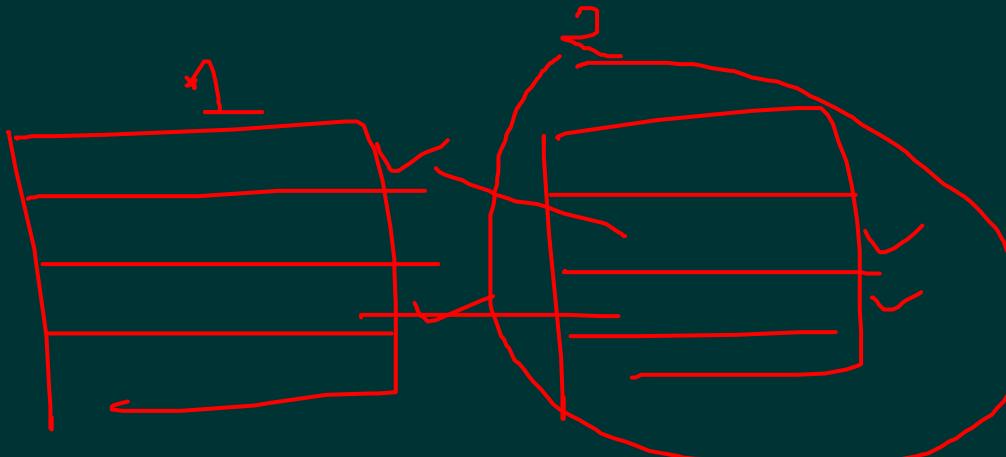
- Join Expressions
- Views
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- Authorization

# 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

# 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.



# Join operations - Example

Relation course

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

Relation prereq

| course_id | prereq_id |
|-----------|-----------|
| BIO-301   | BIO-101   |
| CS-190    | CS-101    |
| CS-347    | CS-101    |

*Observe that*

- o *prereq information is missing for CS-315 and*
- o *course information is missing for CS-437*

# Natural or conditional Joins

Q

*select \* from course natural join prereq*

✓

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

✓

| course_id | prereq_id |
|-----------|-----------|
| BIO-301   | BIO-101   |
| CS-190    | CS-101    |
| CS-347    | CS-101    |

✓

| 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    | R           | C.S        | 3       | Null      |

# Natural or conditional Joins

`select * from course natural join prereq`

| <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              |

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

# Left Outer Join

*select \* from course natural left outer join prereq*

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

| course_id | prereq_id |
|-----------|-----------|
| BIO-301   | BIO-101   |
| CS-190    | CS-101    |
| CS-347    | CS-101    |

# Left Outer Join

*select \* from course natural left outer join prereq*

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

| course_id | prereq_id |
|-----------|-----------|
| BIO-301   | BIO-101   |
| CS-190    | CS-101    |
| CS-347    | CS-101    |

| 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      |

# Right Outer Join

select \* from course **natural right outer join** prereq

| <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              |

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

# Right Outer Join

select \* from course natural right outer join prereq

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

| course_id | prereq_id |
|-----------|-----------|
| BIO-301   | BIO-101   |
| CS-190    | CS-101    |
| CS-347    | CS-101    |

| 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    |

# Full Outer Join

n course **natural full outer join** prereq

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

| course_id | prereq_id |
|-----------|-----------|
| BIO-301   | BIO-101   |
| CS-190    | CS-101    |
| CS-347    | CS-101    |

| 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

- **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, and what attributes are present in the result of the join.
- **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> | <i>Join Conditions</i>           |
|-------------------|----------------------------------|
| <u>inner join</u> |                                  |
| left outer join   |                                  |
| right outer join  |                                  |
| full outer join   |                                  |
|                   | natural                          |
|                   | on <predicate>                   |
|                   | using ( $A_1, A_1, \dots, A_n$ ) |

# 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

select \* from course **natural right 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-347           | <i>null</i>  | <i>null</i>      | <i>null</i>    | CS-101           |

select \* from course **full outer join** prereq **using (course\_id)**

| <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>      |
| CS-347           | <i>null</i>  | <i>null</i>      | <i>null</i>    | 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.

# Example Views

- 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

- **create view** physics\_fall\_2009 **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 =  
'2009';
- **create view** physics\_fall\_2009\_watson **as**  
**select** course\_id, room\_number  
**from** physics\_fall\_2009 **where** building= 'Watson';  
(Physics courses which are held in 'Watson' building during Fall 2009)

# View Expansion

- Expand use of a view in a query/another view

```
create view physics_fall_2009_watson as
(select course_id, room_number
from (select course.course_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 = '2009')
 where building= 'Watson';
```

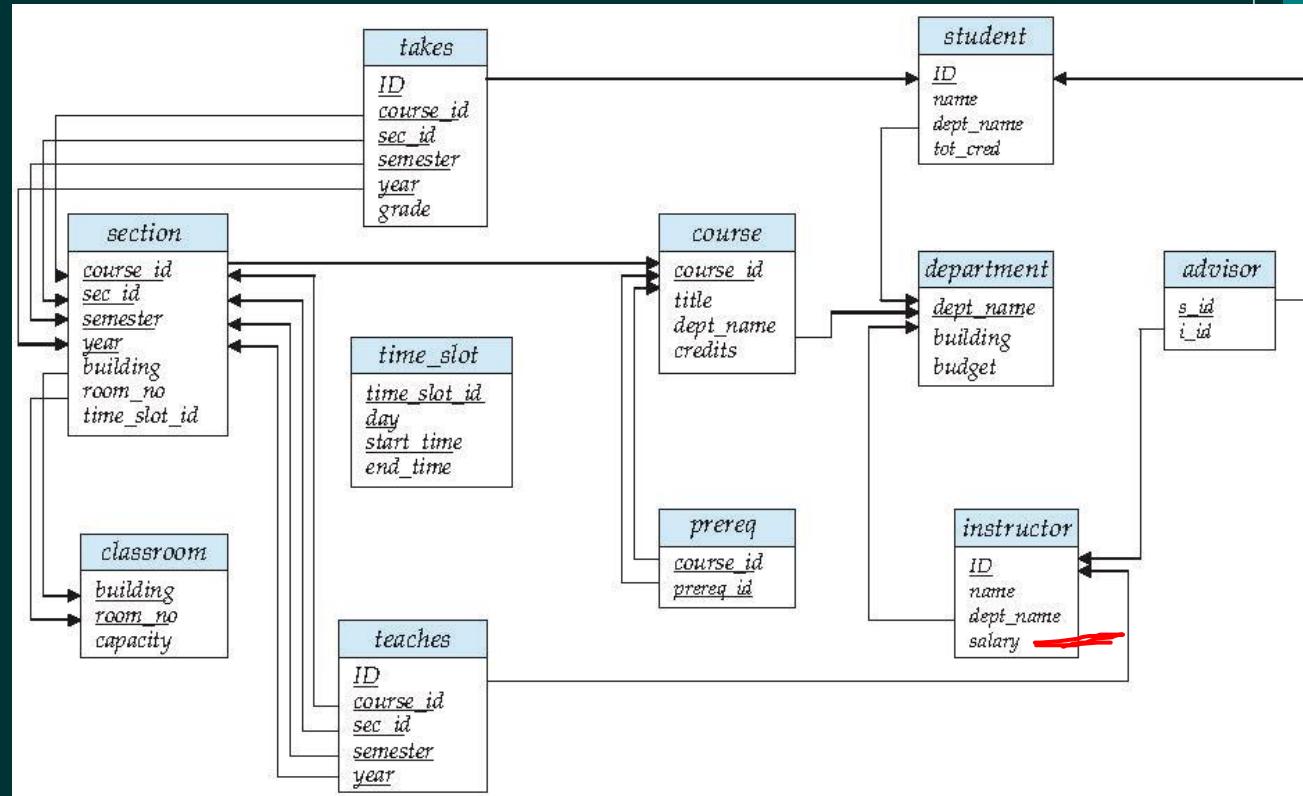
```
create view faculty as select ID, name, dept_name from instructor
```

# Update of a View

- Add a new tuple to *faculty* view which we defined earlier

```
insert into faculty values ('30765', 'Green', 'Music');
```

This insertion must be represented by  
the insertion of the tuple  
('30765', 'Green', 'Music', null)  
into the *instructor* relation



# Some Updates cannot be Translated Uniquely

- **create view** *instructor\_info* **as**  
    **select** *ID, name, building*  
    **from** *instructor, department*  
    **where** *instructor.dept\_name= department.dept\_name*;
- **insert into** *instructor\_info* **values** ('69987', 'White', 'AB5');
  - which department, if multiple departments in AB5?
  - what if no department is in AB5?

- Most SQL implementations allow updates only on simple views
  - The **from** clause has only one database relation. ✓
  - The **select** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or **distinct** specification.
  - Any attribute not listed in the **select** clause can be set to null ↴
  - The query does not have a **group** by or **having** clause.

# And Some Not at All

- **create view** history\_instructors **as**  
select \* from instructor where dept\_name= 'History';
- What happens if we insert ('25566', 'Brown', 'Biology', 100000) into history\_instructors?

No check option , hence allows to insert biology instructor to the table.

**CREATE OR REPLACE VIEW** vps **AS SELECT** employeeNumber, lastName, firstName, jobTitle, extension, email, officeCode, reportsTo **FROM** employees  
**WHERE** jobTitle **LIKE** '%VP%' **WITH CHECK OPTION;**

# Materialized Views

**Materializing a view:** create a physical table containing all the tuples in the result of the query defining the view

If relations used in the query are updated, the materialized view result becomes out of date

- Need to **Maintain** the view, by updating the view whenever the underlying relations are updated.

# Transactions

- Unit of work
- Atomic transaction
  - either fully executed or rolled back as if it never occurred
- Isolation from concurrent transactions
- Transactions begin implicitly
  - Ended by **commit work** or **rollback work**
- But default on most databases: each SQL statement commits automatically
  - Can turn off auto commit for a session (e.g. using API)
  - In SQL:1999, can use: **begin atomic .... end**
    - Not supported on most databases

# 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

# Integrity Constraints on a Single Relation

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

# Not Null and Unique Constraints

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

*name* **varchar**(20) **not null**

*budget* **numeric**(12,2) **not null**

- **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).
  - Eg.: Unique(*Secld*, *Roll\_No*)

# The check clause

**check (P)** : where P is a predicate

Example: ensure that semester is one of fall, winter, spring or summer:

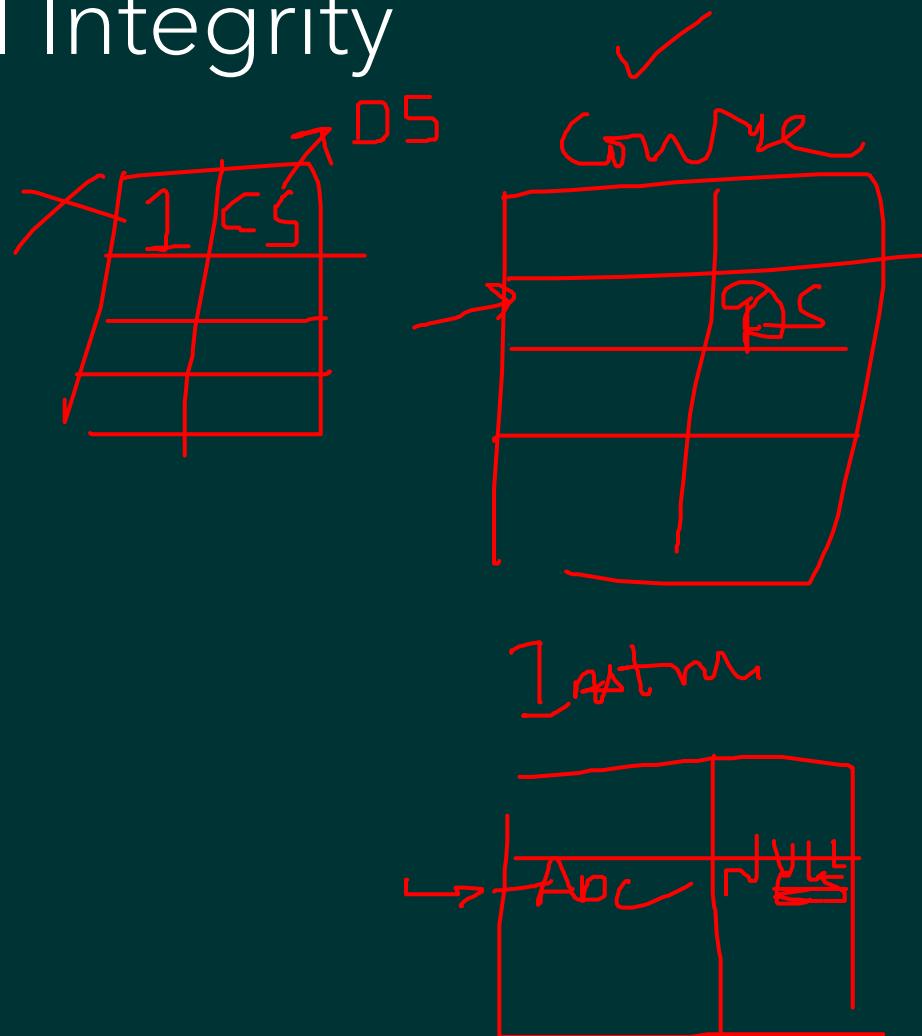
```
create table section (
 course_id varchar (8),
 sec_id varchar (8),
 semester varchar (6),
 year numeric (4,0),
 building varchar (15),
 room_number varchar (7),
 time_slot_id varchar (4),
 primary key (course_id, sec_id, semester, year),
 check (semester in ('Fall', 'Winter', 'Spring', 'Summer')));
```

# 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.

# Cascading Actions in Referential Integrity

- **create table** course (  
    course\_id **char(5) primary key**,  
    title **varchar(20)**,  
    dept\_name **varchar(20) references department**  
)
- **create table** course (  
    ...  
    dept\_name **varchar(20)**,  
    **foreign key** (dept\_name) **references department**  
        **on delete cascade** ✓  
        **on update cascade**, ✓  
    ...  
)
- alternative actions to cascade: **set null, set default**



# Complex Check Clauses

- Every section has at least one instructor teaching the section.
  - how to write this?
- Unfortunately: subquery in check clause not supported by pretty much any database
  - Alternative: triggers
- **create assertion** <assertion-name> **check** <predicate>;
  - Also not supported by most of the DBMS
  - Make sure that supervisee's salary is less than his Supervisor's salary

Create **assertion** Emp\_sal\_chek

↙ **Check (not exists** (select \* from Employee E1, Employee E\_Mgr where E1.Mgr\_Id= E\_Mgr.Emp\_Id and E1.salary> E\_Mgr.salasry) );

# 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** *Rupees* **as** numeric(12,2)

- **create table** *department3* (*dept\_name* **varchar** (20), *building* **varchar** (15),  
*budget Rupees*);
- Create table Instructor (*ID* **varchar**(3), *name* **varchar**(10), *Salary Rupees* );

# 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**, **check** specified on them.

**create domain** degree\_level **varchar(10)**  
**constraint** degree\_level\_test  
**check (value in** ('Bachelors', 'Masters', 'Doctorate'));

- Create table Instructor (ID varchar(3), name *person\_name* , Salary Rupees , Qualification *degree\_level* );

# Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:
- **blob**: binary large object -- object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
- **clob**: character large object -- object is a large collection of character data
- When a query returns a large object, a pointer is returned rather than the large object itself.

# Authorization

Forms of authorization 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.

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 name or view name> **to** <user list>

- <user list> is:

- a user-id
- **public**, which allows all valid users the privilege granted
- A role (more on this later)
- 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 name or view name> **from** <user list>

- Example:

**revoke select on branch from**  $U_1, U_2, U_3$

- <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 grantors, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being revoked are also revoked.

# Roles

- **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;

Ex:

```
REVOKE privilege_name
ON object_name
FROM {user_name | PUBLIC | role_name}
```

# 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**;
- Etc. read Section 4.6 for more details we have omitted here.



# Chapter 15 : Concurrency Control

Database System Concepts, 6<sup>th</sup> Ed.

©Silberschatz, Korth and Sudarshan

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



# Chapter 15: Concurrency Control

- Lock-Based Protocols
- Timestamp-Based Protocols
- Validation-Based Protocols



# Concurrent Schedules

- Advantage of Concurrent Schedules
- Problem associated with the execution of concurrent schedules
  - Lost Update
  - Dirty Read
  - Incorrect Summary



# Concurrency Protocols

Goal: Generates Concurrent Schedules

Should satisfy following facts:

- Schedule is **serializable** : Conflict or View serializable
- Recoverable
- Cascade-less
- Increase the level of **concurrency** (To improve system performance)
- Reduce the protocol **overhead**



# Lock based protocol

## Lock Mechanism

*exclusive (X) for Write operation*

*shared (S) for Read operation*

*Concurrency control Manager*

lock-X instruction

lock-S instruction



# Lock-Based Protocols

- A **lock** is a mechanism to control concurrent access to a data item
- It is a variable associated with a data item in the database and describes the status of the item w.r.t. possible operations that can be applied on to item.
- Generally there is **one lock for each item**.
- **Binary Locks**: can have two states or values: **locked and unlocked** (1 and 0)
- Data items can be locked in two modes :
  1. **exclusive (X) mode**. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  2. **shared (S) mode**. Data item can only be read. S-lock is requested using **lock-S** instruction.
- Lock requests are made to **concurrency-control manager**. Transaction can proceed only after request is granted.



# Lock-Based Protocols (Cont.)

## □ Lock-compatibility matrix

|              | S     | X     |
|--------------|-------|-------|
| <del>S</del> | true  | false |
| X            | false | false |

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item,
  - but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.



# Lock-Based Protocols (Cont.)

- Example of a transaction performing locking:

$T_2$ : **lock-S(A);**  
~~**read (A);**~~  
~~**unlock(A);**~~  
**lock-S(B);**  
~~**read (B);**~~  
**unlock(B);**  
**display(A+B)**

T1

*May lead to incorrect Summary Problem*

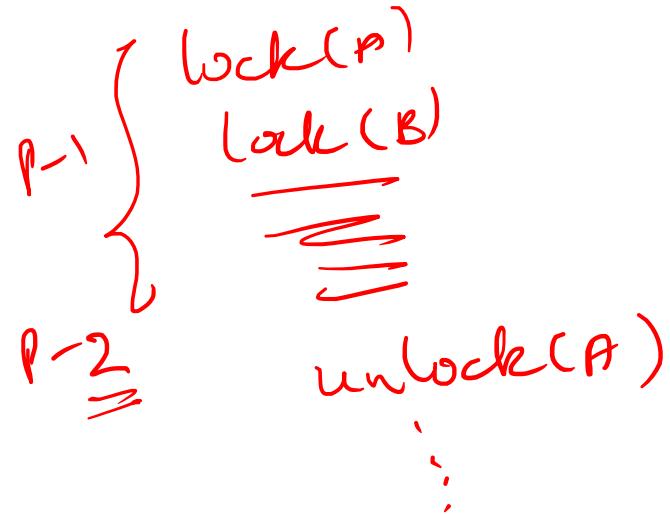
- Locking as above is **not sufficient to guarantee serializability** — if  $A$  and  $B$  get updated in-between the read of  $A$  and  $B$ , the displayed sum would be wrong.
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.



# The Two-Phase Locking Protocol

~~SQL~~

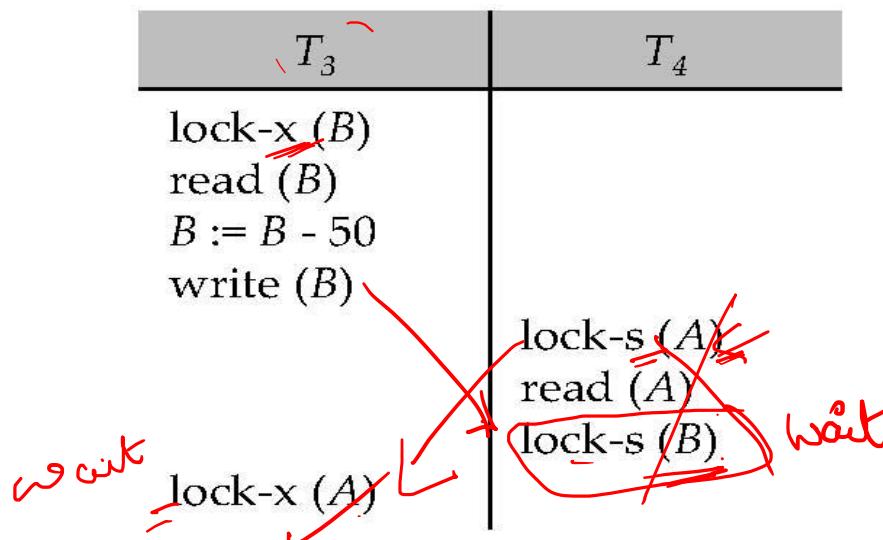
- This is a protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: Shrinking Phase
  - transaction may release locks
  - transaction may not obtain locks
- The protocol assures serializability.
- It can be proved that the transactions can be serialized in the order of their lock points (i.e. the point where a transaction acquired its final lock).





# Pitfalls of Lock-Based Protocols

- Consider the partial schedule



- Neither  $T_3$  nor  $T_4$  can make progress — executing **lock-x( $B$ )** causes  $T_4$  to wait for  $T_3$  to release its lock on  $B$ , while executing **lock-X( $A$ )** causes  $T_3$  to wait for  $T_4$  to release its lock on  $A$ .
- Such a situation is called a **deadlock**.
  - To handle a deadlock **one of  $T_3$  or  $T_4$  must be rolled back** and its **locks released**.

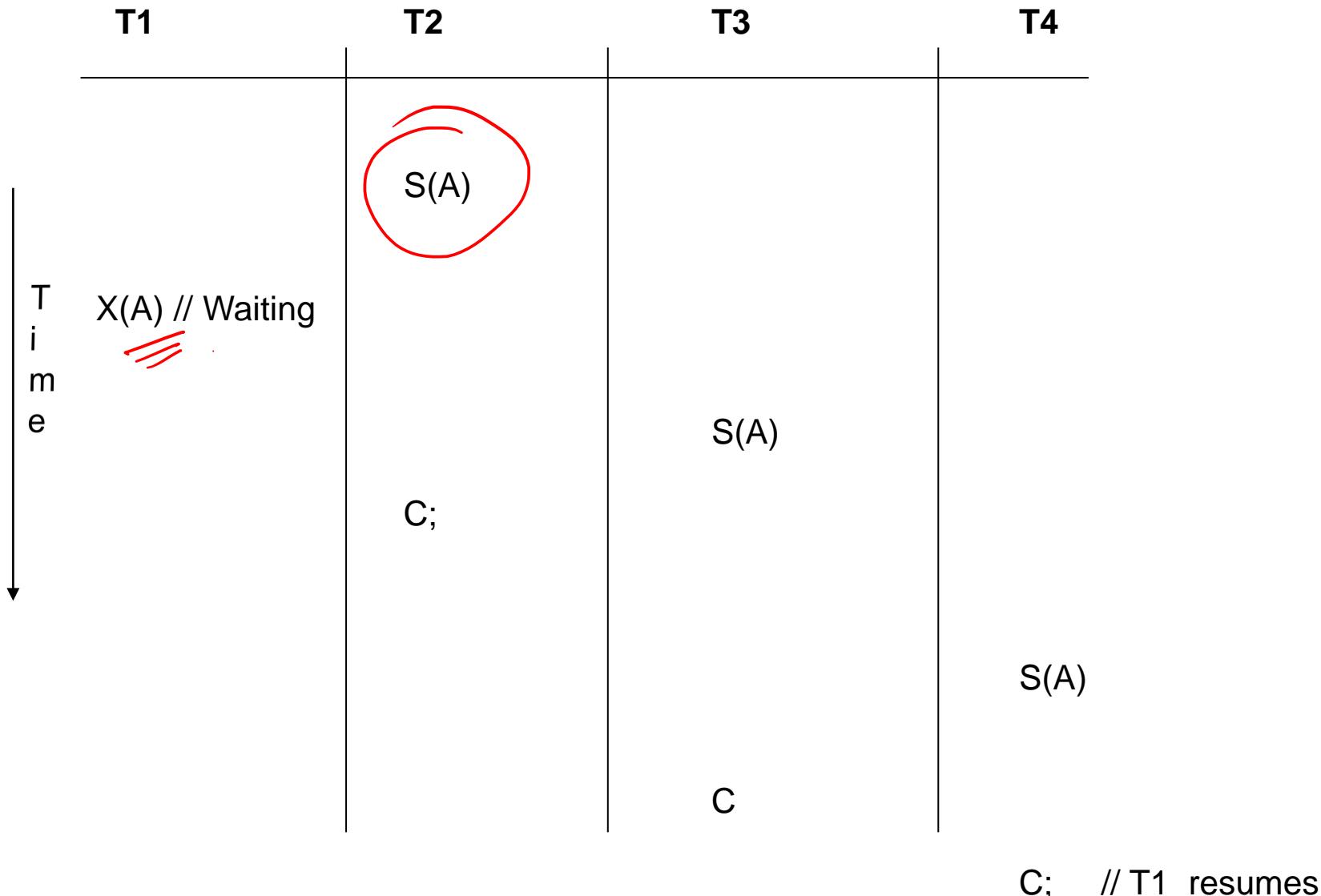


# Pitfalls of Lock-Based Protocols (Cont.)

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a **sequence of other transactions** request and are granted an S-lock on the same item.
  - The same transaction is **repeatedly rolled back due to deadlocks**.
- Concurrency control manager can be designed to **prevent starvation**.



# Example for Starvation





# The Two-Phase Locking Protocol (Cont.)

- Cascading roll-back is possible under two-phase locking.

— T1                    T2

X(A)

UN(A)

X(A)

Aborts

S 2 P L

locks  
:  
:  
Commit

unlocks

- Solution: strict two-phase locking:

□ Here a transaction must hold all its exclusive locks till it commits/aborts. After commit it must unlock.

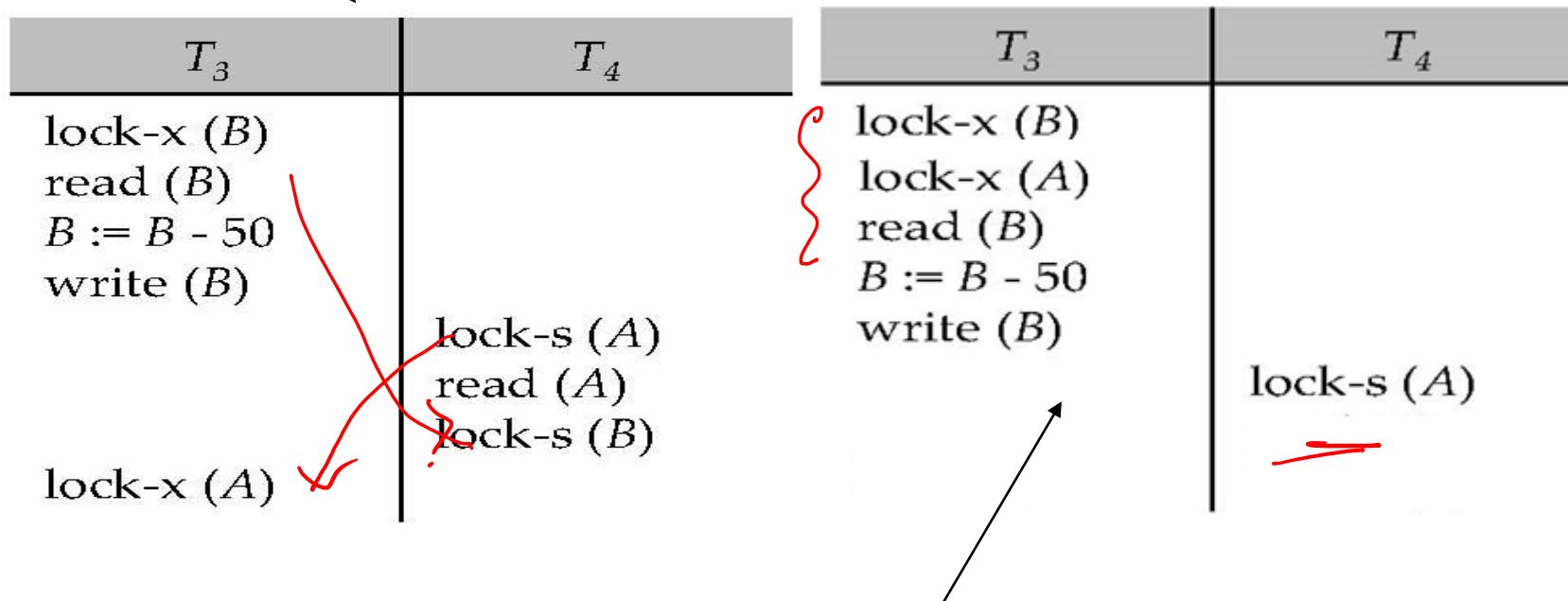
X locks  
X and S

- Rigorous two-phase locking is even stricter: here all locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.



# Static/ Conservative 2PL protocol

## ○ Deadlock:



- Solution: Static/ Conservative 2PL protocol
  - Gets Lock in **Atomic Manner**
- No Deadlock
- Reduces the concurrency level



# Lock Conversions

To improve the concurrency level

2PL + Lock  
Conversion

- Two-phase locking with lock conversions:

- First Phase:

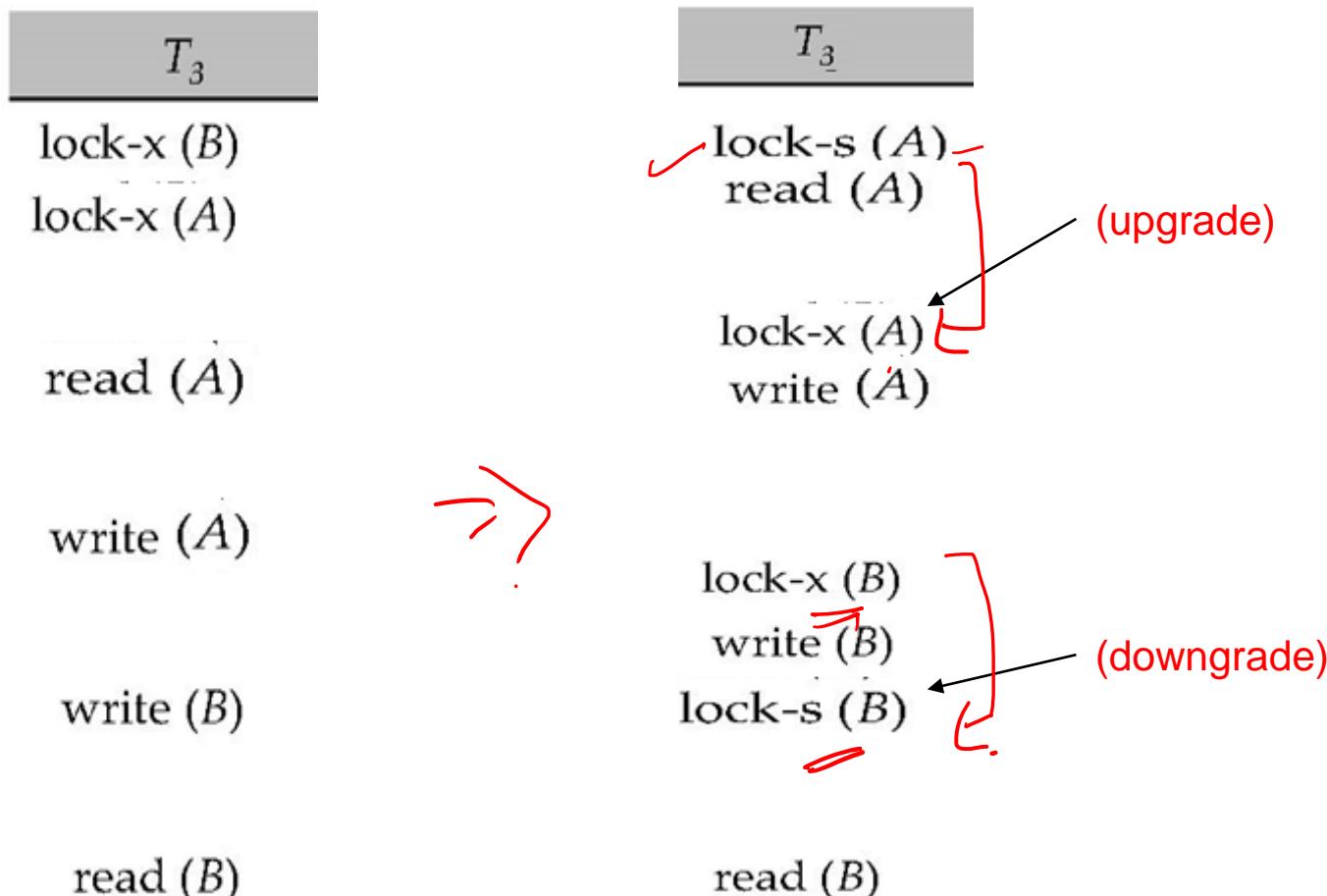
- can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade)

- Second Phase:

- can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (downgrade)



# Lock Conversions





# Automatic Acquisition of Locks

- A transaction  $T_i$  issues the standard read/write instruction, without explicit locking calls.
- The operation **read( $D$ )** is processed as:

```
if T_i has a lock on D
 then
 read(D)
 else begin
 if necessary wait until no other
 transaction has a lock-X on D
 grant T_i a lock-S on D ;
 read(D)
 end
```



# Automatic Acquisition of Locks (Cont.)

- **write( $D$ )** is processed as:

if  $T_i$  has a **lock-X** on  $D$

**then**

write( $D$ )

**else begin**

if necessary wait until no other trans. has any lock on  $D$ ,

if  $T_i$  has a **lock-S** on  $D$

**then**

upgrade lock on  $D$  to **lock-X**

**else**

grant  $T_i$  a **lock-X** on  $D$

write( $D$ )

**end;**

- All locks are released after commit or abort



# Which transaction implements rigorous 2PL?

(A)

strict 2PL  
rigorous 2PL

| <i>T<sub>1</sub></i> |
|----------------------|
| lock-X(A)            |
| read(A)              |
| write(A)             |
| lock-S(B)            |
| read(B)              |
| unlock(A)            |
| unlock(B)            |
| commit               |

2PL

| <i>T<sub>2</sub></i> |
|----------------------|
| lock-X(A)            |
| read(A)              |
| write(A)             |
| lock-S(B)            |
| read(B)              |
| commit               |
| unlock(A)            |
| unlock(B)            |

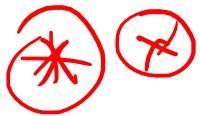
rigorous 2PL

*T<sub>2</sub>*

lock X(A)  
R(A)  
W(A)  
lock S(B)  
R(B)  
unlock(B)  
commit

unlock X(A)  
strict 2PL =





# Deadlock Handling

- Schedule with deadlock

$T_1$  waits  $T_2$

$T_2$  waits  $T_1$

| $T_1$                                                                       | $T_2$                                                                       |
|-----------------------------------------------------------------------------|-----------------------------------------------------------------------------|
| <del>lock-X on A<br/>write (A)</del><br><br><del>wait for lock-X on B</del> | <del>lock-X on B<br/>write (B)</del><br><br><del>wait for lock-X on A</del> |

solution:

One of the transactions has to be aborted (rollback) to release certain locks.



No deadlock

| $S_1$     |           |
|-----------|-----------|
| $T_1$     | $T_2$     |
| lock-X(A) |           |
| read(A)   |           |
| write(A)  |           |
| unlock(A) |           |
|           | lock-S(B) |
|           | read(B)   |
| lock-S(B) |           |
| read(B)   |           |
| unlock(B) |           |
|           | lock-X(A) |
|           | read(A)   |
|           | write(A)  |
|           | unlock(A) |
|           | unlock(B) |

Cycles  $\Rightarrow$  Deadlocks  
are seen.

| $S_2$     |           |
|-----------|-----------|
| $T_1$     | $T_2$     |
| lock-X(A) |           |
| read(A)   | wait      |
| write(A)  |           |
|           | lock-X(B) |
|           | read(B)   |
| lock-S(B) | wait      |
|           | lock-X(A) |
| read(B)   | read(A)   |
| unlock(B) | write(A)  |
| unlock(A) |           |
|           | unlock(A) |
|           | unlock(B) |



# Deadlock Handling

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- **Deadlock prevention** protocols ensure that the **system will never enter into a deadlock state.**
- Some prevention strategies :
  - Require that each transaction locks all its data items before it begins execution (predeclaration or conservative two phase locking).
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol)



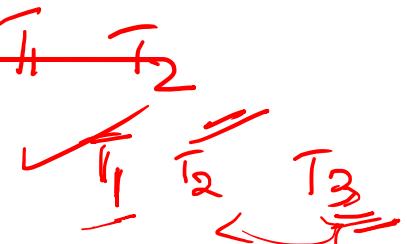
# More Deadlock Prevention Strategies

- Following schemes use **transaction timestamps** for the sake of deadlock prevention alone.  
*older ✓ younger*
- **wait-die** scheme — non-preemptive
  - If  $TS(T_i) < TS(T_j)$  then  $T_i$  is allowed to wait otherwise abort  $T_i$  and restart it later with the same timestamp.
  - Older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead.
  - a transaction may die several times before acquiring needed data item
- **wound-wait** scheme — preemptive
  - If  $TS(T_i) < TS(T_j)$  then abort  $T_i$  and restart it later with the same timestamp otherwise  $T_i$  is allowed to wait
  - older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.

$$\begin{aligned} T_i &= @ \text{ 100 } \text{ older} \\ T_j &@ \text{ 150 } \text{ younger} \end{aligned}$$



Suppose in a database, there are three transactions  $T_1$ ,  $T_2$  and  $T_3$  with timestamp 10, 20 and 30 respectively.  $T_2$  is holding a data item which  $T_1$  and  $T_3$  are requesting to acquire in respect of Wound-Wait Deadlock Prevention scheme, what is the outcome?



$T_1$  will continue

Transaction  $T_2$  will be aborted. Transaction  $T_3$  will wait for  $T_2$  to release the data item.

in respect of Wait-Die Deadlock Prevention scheme, what is the outcome?

$T_2$  continues

Transaction  $T_1$  will wait for  $T_2$  to release data item.

Transaction  $T_3$  will die

/abort

$T_1$  wait       $T_2$  dies  
 $T_3$  die



# Deadlock prevention (Cont.)

- Both in *wait-die* and in *wound-wait* schemes, a **rolled back transactions is restarted with its original timestamp**. Older transactions thus have precedence over newer ones, and starvation is hence avoided.
- **Timeout-Based Schemes:**
  - a transaction waits for a **lock only for a specified amount of time**. After that, the wait times out and the transaction is rolled back.
  - thus deadlocks are not possible
  - simple to implement; but **starvation** is possible. Also difficult to determine good value of the timeout interval.

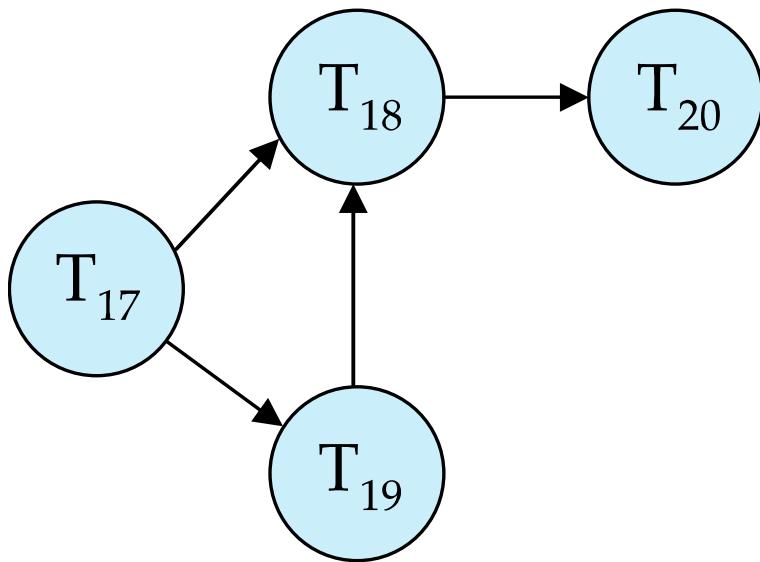


# Deadlock Detection

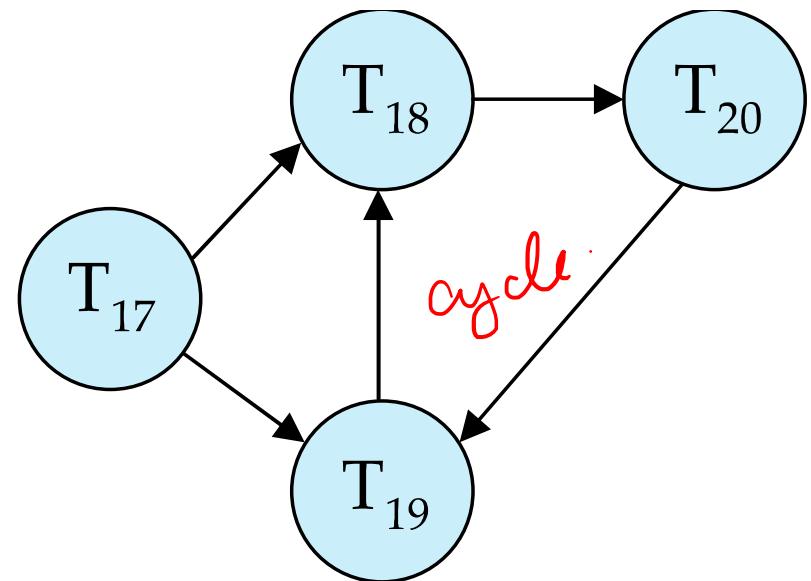
- Deadlock Detection where we periodically check to see if the system is in a state of deadlock. This can happen if the transaction weight is less and so they repeatedly lock the item.
- Deadlocks can be described as **a wait-for graph**, which consists of a pair  $G = (V, E)$ ,
  - $V$  is a set of vertices (all the **transactions** in the system)
  - $E$  is a set of edges; each element is an ordered pair  $T_i \rightarrow T_j$ .
- If  $T_i \rightarrow T_j$  is in  $E$ , then there is a directed edge from  $T_i$  to  $T_j$ , implying that  $T_i$  is waiting for  $T_j$  to release a data item.
- When  $T_i$  requests a data item currently being held by  $T_j$ , then the edge  $T_i \rightarrow T_j$  is inserted in the wait-for graph. This edge is removed only when  $T_j$  is no longer holding a data item needed by  $T_i$ .
- The **system is in a deadlock state** if and only if the **wait-for graph has a cycle**. Must invoke a deadlock-detection algorithm periodically to look for cycles.



# Deadlock Detection (Cont.)



Wait-for graph without a cycle



Wait-for graph with a cycle



# Deadlock Recovery

- When deadlock is detected :
  - Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as **victim that will incur minimum cost**.
  - Rollback -- determine how far to roll back transaction
    - ▶ **Total rollback:** Abort the transaction and then **restart it**.
    - ▶ More effective to roll back transaction only as far **as necessary to break deadlock**.
  - **Starvation** happens if same transaction is always chosen as victim. Include the **number of rollbacks** in the cost factor **to avoid starvation**



# Timestamp-Based Protocols

- Each transaction is issued a **timestamp** when it enters the system. If an old transaction  $T_i$  has time-stamp  $\underline{\text{TS}}(T_i)$ , a new transaction  $T_j$  is assigned time-stamp  $\text{TS}(T_j)$  such that  $\text{TS}(T_i) < \underline{\text{TS}}(T_j)$ .  

- The protocol manages concurrent execution such that the **time-stamps** determine the **serializability order**.
- In order to assure such behavior, the protocol maintains for each data 'Q' two timestamp values:
  - W-timestamp(Q) is the **largest time-stamp (Youngest Transaction)** of any transaction that executed write(Q) successfully.
  - R-timestamp(Q) is the **largest time-stamp** of any transaction that executed read(Q) successfully.

□

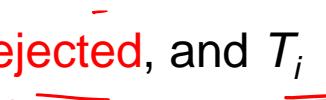


# Timestamp-Based Protocols (Cont.)

X K

- The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order.  

- Suppose a transaction  $T_i$  issues a read( $Q$ )—
  1. If  $TS(T_i) \leq W\text{-timestamp}(Q)$ , then  $T_i$  needs to read a value of  $Q$  that was already overwritten.  

  2. “ $Q$  written by younger  $T$ ” read by older  $T$ 
    - Hence, the **read** operation is rejected, and  $T_i$  is rolled back.  

  3. If  $TS(T_i) \geq W\text{-timestamp}(Q)$ , then the **read** operation is executed, and  $R\text{-timestamp}(Q)$  is set to  $\max(R\text{-timestamp}(Q), TS(T_i))$ .  

  4.  $Q$  written by older  $T$



# Timestamp-Based Protocols (Cont.)

- Suppose that transaction  $T_i$  issues write(Q).

1. If  $\text{TS}(T_i) < \underline{\text{R-timestamp}}(Q)$ , then the value of Q that  $T_i$  is producing was needed previously, and the system assumed that that value would never be produced.
  - Hence, the write operation is rejected, and  $T_i$  is rolled back.
2. Q read by younger T that is written by older T
  - Hence, the write operation is rejected, and  $T_i$  is rolled back.
3. If  $\text{TS}(T_i) < \underline{\text{W-timestamp}}(Q)$ , then  $T_i$  is attempting to write an obsolete value of Q.
  - Hence, this write operation is rejected, and  $T_i$  is rolled back.
4. Otherwise, the write operation is executed, and  $\text{W-timestamp}(Q)$  is set to  $\text{TS}(T_i)$ .



older transaction has to commit

$T_1$  (read)

| 10                  | 20                  |
|---------------------|---------------------|
| older<br>$T_1$      | younger<br>$T_2$    |
| $r(q)$<br>$q = 100$ | $w(q)$<br>$q = 100$ |
| $r(q)$<br>$q = 100$ | fails               |

$\bar{T}_1$  (write)

| 10                  | 20       |
|---------------------|----------|
| $T_1$               | $T_2$    |
| $R(q)$              | $q = 50$ |
| $W(q)$<br>$q = 100$ | fails    |

$R_0$  (back)

commit

$\bar{T}_2$  (write)

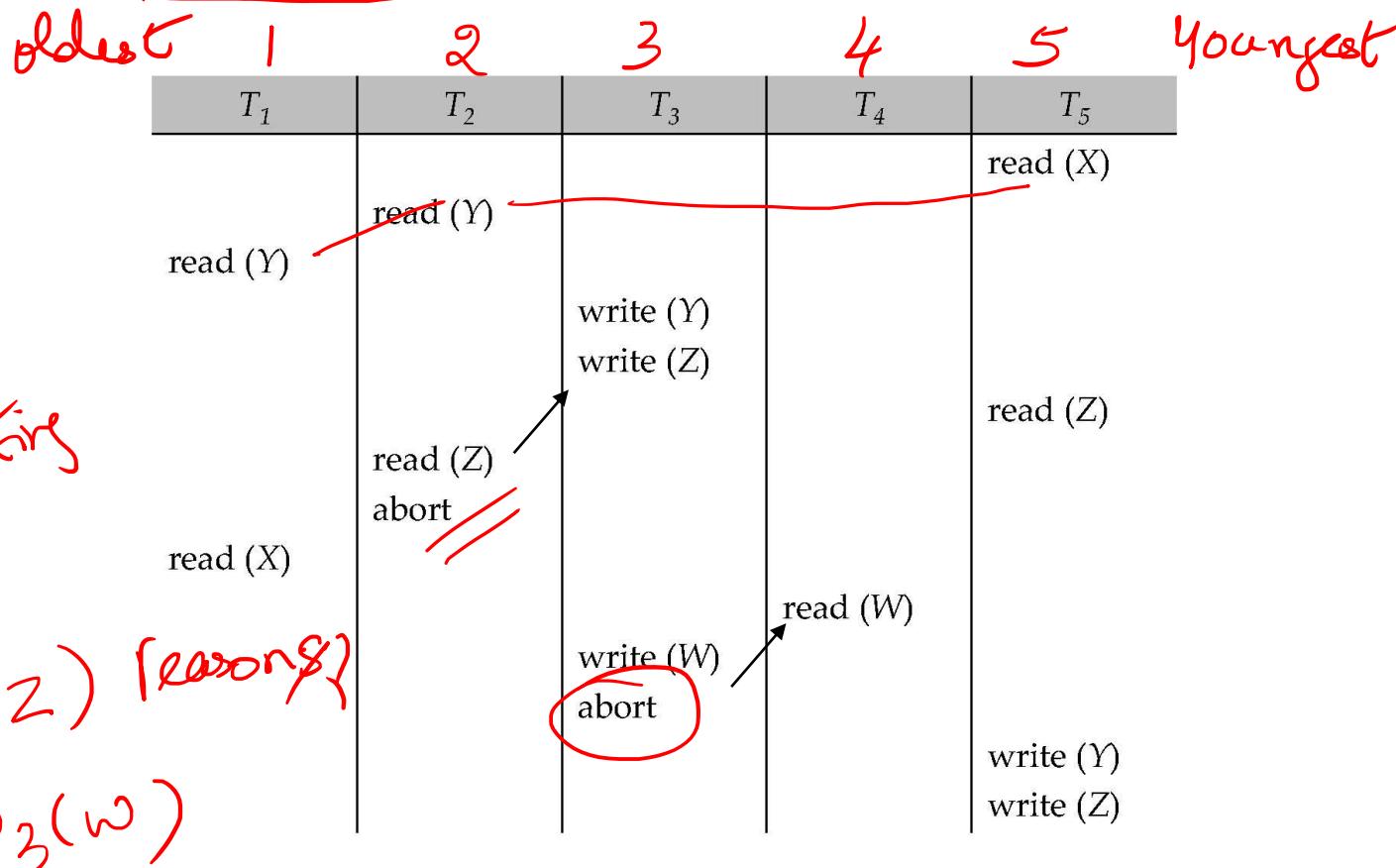
| 10                  | 20       |
|---------------------|----------|
| $T_1$               | $T_2$    |
| $w(q)$<br>$q = 50$  | $q = 2?$ |
| $W(q)$<br>$q = 100$ |          |

$R_0$  (back)



# Example Use of the Protocol

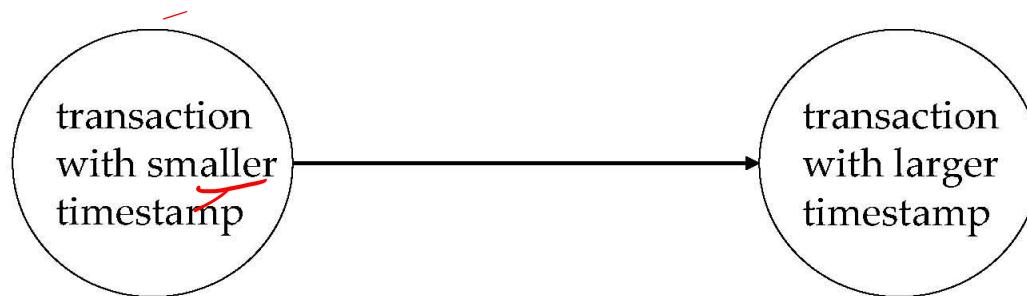
A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5. Apply Timestamp based protocol.





# Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol **guarantees serializability** since all the arcs in the **precedence graph** are of the form:



Thus, there will be **no cycles** in the precedence graph

- Timestamp protocol **ensures freedom from deadlock** as no transaction ever waits.
- But the schedule may **not be cascade-free**, and **may not even be recoverable**: Due to absence of Locking mechanism



# Validation-Based Protocol

- Execution of transaction  $T_i$  is done in **three phases**.
  1. **Read and execution phase**: Transaction  $T_i$  writes only to temporary local variables
  2. **Validation phase**: Transaction  $T_i$  performs a ``validation test'' to determine if local variables can be written without violating serializability. *conflict free*.
  3. **Write phase**: If  $T_i$  is validated, the updates are applied to the database; otherwise,  $T_i$  is rolled back.
- The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.
  - Assume for simplicity that the validation and write phase occur together, atomically and serially
    - ▶ I.e., only one transaction executes validation/write at a time.
  - Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation



# Validation-Based Protocol (Cont.)

- Each transaction  $T_i$  has 3 timestamps
  - $\text{Start}(T_i)$  : the time when  $T_i$  started its execution
  - $\text{Validation}(T_i)$ : the time when  $T_i$  entered its validation phase
  - $\text{Finish}(T_i)$  : the time when  $T_i$  finished its write phase \_\_\_\_\_
- **Serializability order** is determined by timestamp given at validation time, to increase concurrency.
  - Thus  $\text{TS}(T_i)$  is given the value of  $\text{Validation}(T_i)$ .



# Validation Test for Transaction $T_j$

- To handle Read and Write phases clash
- If for all  $T_i$  with  $\text{TS}(T_i) < \text{TS}(T_j)$  either one of the following condition holds:
  1.  $\text{finish}(T_i) < \text{start}(T_j)$  T<sub>j</sub>'s validation test.
  2.  $\text{finish}(T_i) < \text{validation}(T_j)$  and the set of data items written by  $T_i$  does not intersect with the set of data items read by  $T_j$

then validation succeeds and  $T_j$  can be committed.

Otherwise, validation fails and  $T_j$  is aborted.

- *Justification:* Either the first condition is satisfied, and there is no overlapped execution, or the second condition is satisfied and
  - the writes of  $T_i$  do not affect reads of  $T_j$  since they occur after  $T_i$  has finished its reads.
  - the writes of  $T_i$  do not affect reads of  $T_j$  since  $T_j$  does not read any item written by  $T_i$ .



# Validation protocol

- Cascade-less : Since Validation test make sure that Ti reads only the committed values
- No deadlock : Since make use of Timestamp
- Suffer from Starvation  
When validation test fails.



# Schedule Produced by Validation

- Example of schedule produced using validation

| $T_{25}$                                            | $T_{26}$                                                |
|-----------------------------------------------------|---------------------------------------------------------|
| read ( $B$ )                                        |                                                         |
|                                                     | read ( $B$ )<br>$B := B - 50$                           |
| read ( $A$ )<br>⟨ validate ⟩<br>display ( $A + B$ ) | read ( $A$ )<br>$A := A + 50$                           |
| = =                                                 | ⟨ validate ⟩<br>write ( $B$ )<br>write ( $A$ )<br>finis |

*validity test*      *validation test is passed.*



# End of Chapter

Thanks to Alan Fekete and Sudhir Jorwekar for Snapshot  
Isolation examples

**Database System Concepts, 6<sup>th</sup> Ed.**

©Silberschatz, Korth and Sudarshan

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



# Figure 15.01

|   | S     | X     |
|---|-------|-------|
| S | true  | false |
| X | false | false |



# Figure 15.04

| $T_1$                                                                                  | $T_2$                                                                                                                               | concurrency-control manager                                                                              |
|----------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------|
| lock-x ( $B$ )<br><br>read ( $B$ )<br>$B := B - 50$<br>write ( $B$ )<br>unlock ( $B$ ) | lock-s ( $A$ )<br><br>read ( $A$ )<br>unlock ( $A$ )<br>lock-s ( $B$ )<br><br>read ( $B$ )<br>unlock ( $B$ )<br>display ( $A + B$ ) | grant-x ( $B, T_1$ )<br><br>grant-s ( $A, T_2$ )<br><br>grant-s ( $B, T_2$ )<br><br>grant-x ( $A, T_2$ ) |
| lock-x ( $A$ )<br><br>read ( $A$ )<br>$A := A + 50$<br>write ( $A$ )<br>unlock ( $A$ ) |                                                                                                                                     |                                                                                                          |



# Figure 15.07

| $T_3$                                                                                  | $T_4$                                            |
|----------------------------------------------------------------------------------------|--------------------------------------------------|
| lock-x ( $B$ )<br>read ( $B$ )<br>$B := B - 50$<br>write ( $B$ )<br><br>lock-x ( $A$ ) | lock-s ( $A$ )<br>read ( $A$ )<br>lock-s ( $B$ ) |



# Figure 15.08

| $T_5$                                                                                               | $T_6$                                                             | $T_7$                          |
|-----------------------------------------------------------------------------------------------------|-------------------------------------------------------------------|--------------------------------|
| lock-x ( $A$ )<br>read ( $A$ )<br>lock-s ( $B$ )<br>read ( $B$ )<br>write ( $A$ )<br>unlock ( $A$ ) | lock-x ( $A$ )<br>read ( $A$ )<br>write ( $A$ )<br>unlock ( $A$ ) | lock-s ( $A$ )<br>read ( $A$ ) |
|                                                                                                     |                                                                   |                                |

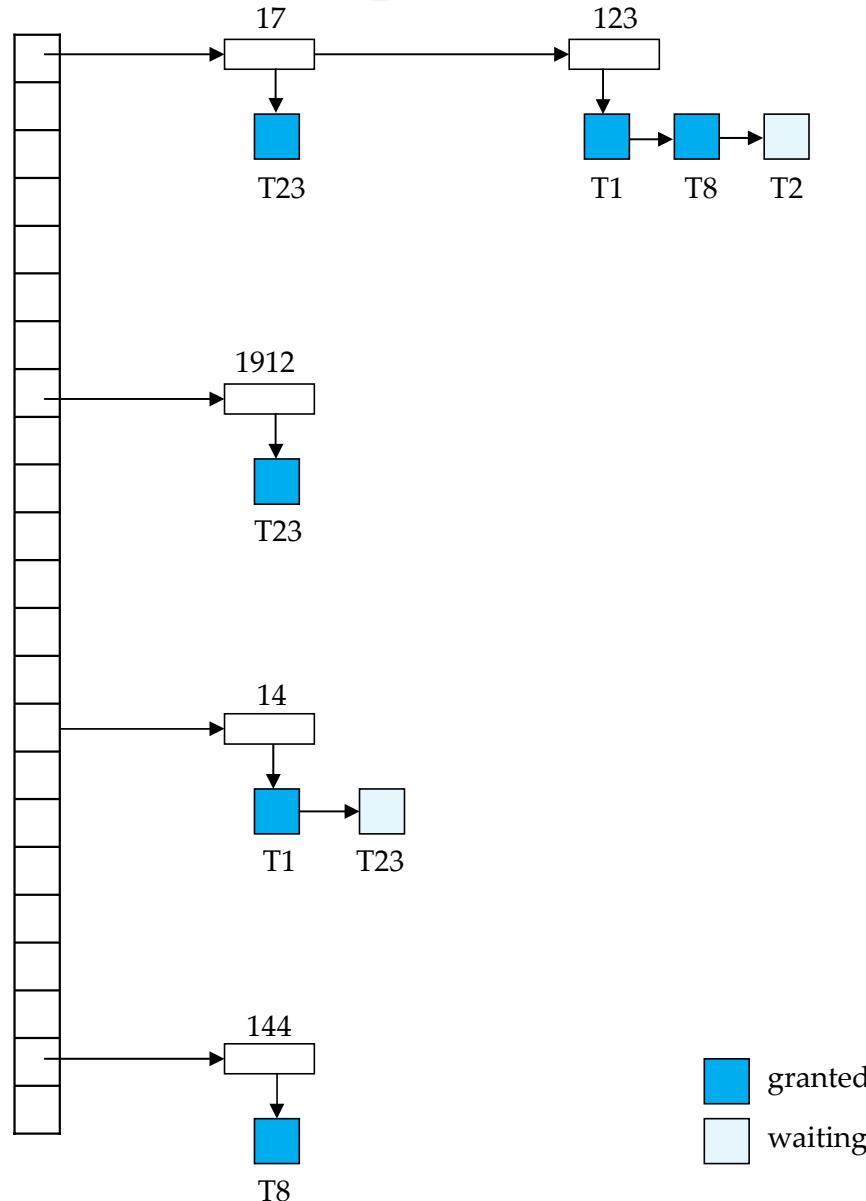


# Figure 15.09

| $T_8$             | $T_9$              |
|-------------------|--------------------|
| lock-s ( $a_1$ )  | lock-s ( $a_1$ )   |
| lock-s ( $a_2$ )  | lock-s ( $a_2$ )   |
| lock-s ( $a_3$ )  | unlock-s ( $a_3$ ) |
| lock-s ( $a_4$ )  | unlock-s ( $a_4$ ) |
| lock-s ( $a_n$ )  |                    |
| upgrade ( $a_1$ ) |                    |

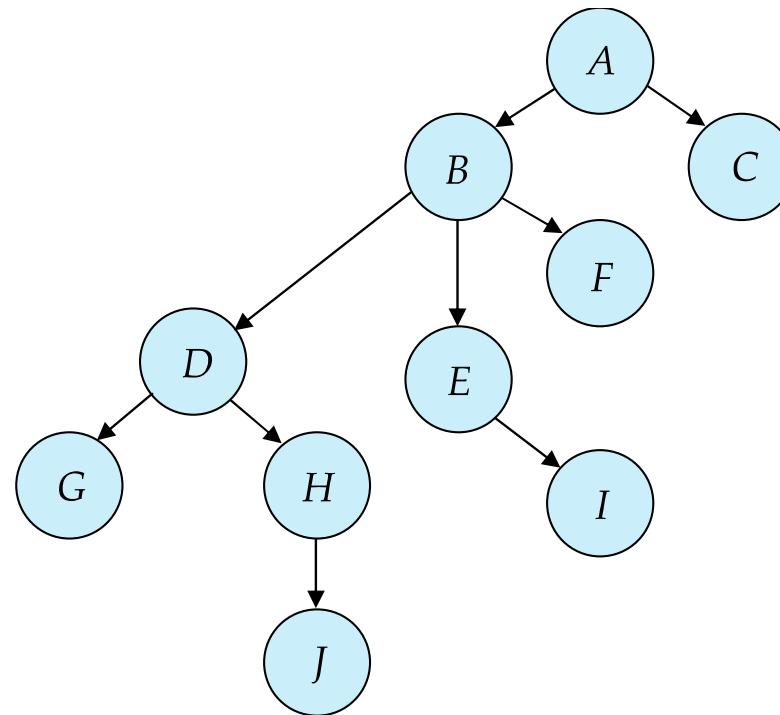


# Figure 15.10





# Figure 15.11



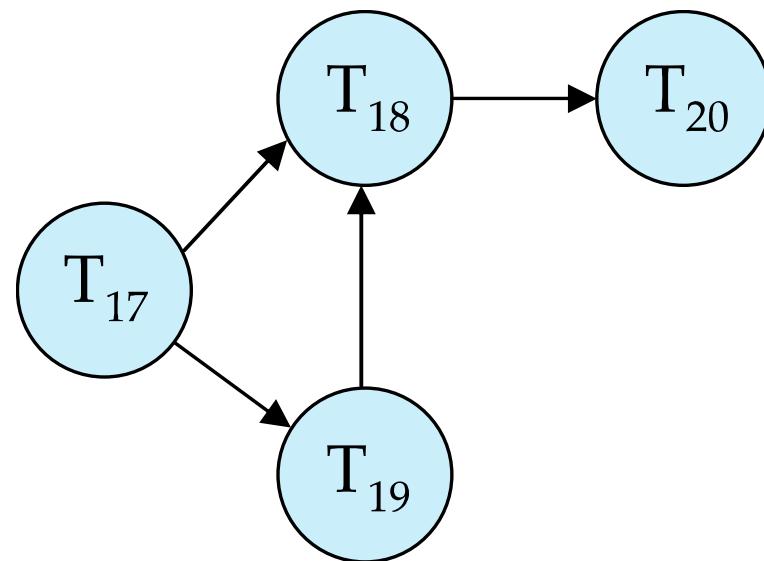


# Figure 15.12

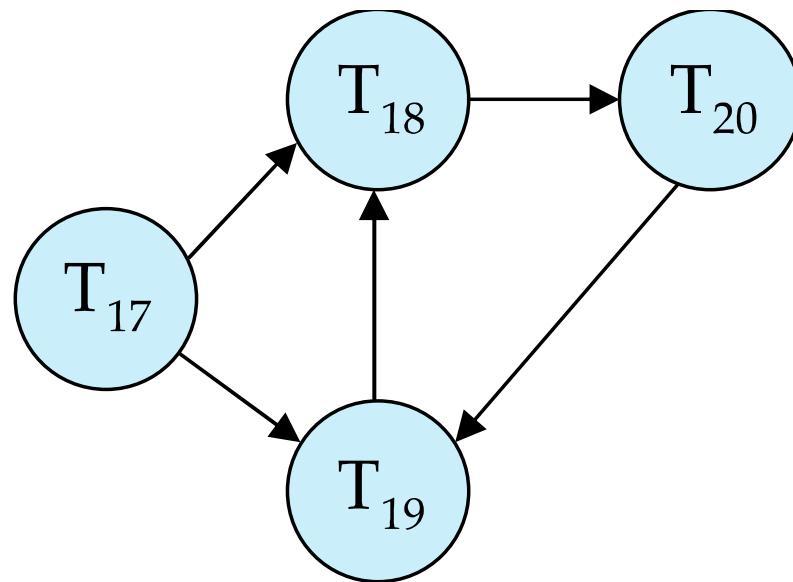
| T <sub>10</sub>                                      | T <sub>11</sub>                        | T <sub>12</sub>          | T <sub>13</sub>                                      |
|------------------------------------------------------|----------------------------------------|--------------------------|------------------------------------------------------|
| lock-x (B)                                           | lock-x (D)<br>lock-x (H)<br>unlock (D) |                          |                                                      |
| lock-x (E)<br>lock-x (D)<br>unlock (B)<br>unlock (E) |                                        | lock-x (B)<br>lock-x (E) |                                                      |
| lock-x (G)<br>unlock (D)                             | unlock (H)                             |                          | lock-x (D)<br>lock-x (H)<br>unlock (D)<br>unlock (H) |
| unlock (G)                                           |                                        | unlock (E)<br>unlock (B) |                                                      |



# Figure 15.13

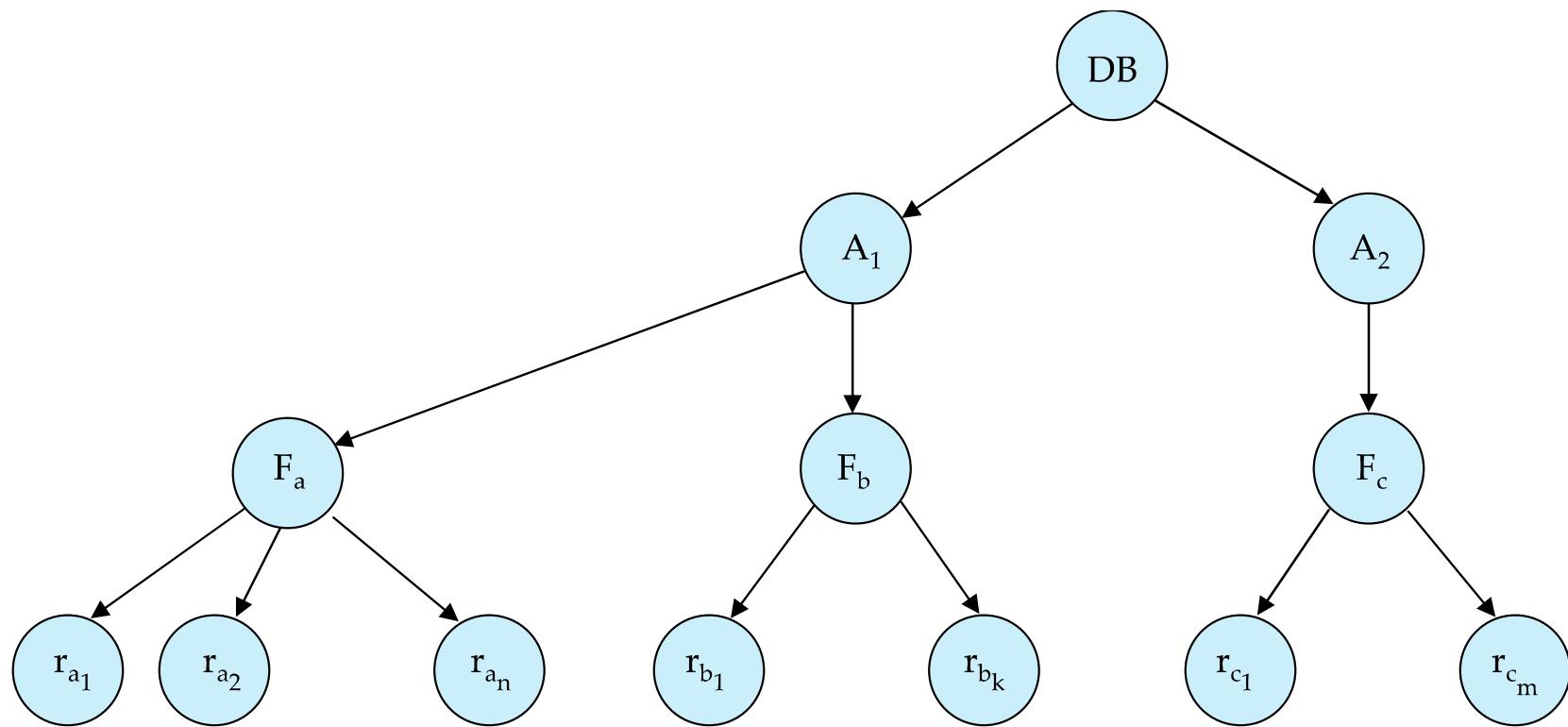


# Figure 15.14





# Figure 15.15





# Figure 15.16

|     | IS    | IX    | S     | SIX   | X     |
|-----|-------|-------|-------|-------|-------|
| IS  | true  | true  | true  | true  | false |
| IX  | true  | true  | false | false | false |
| S   | true  | false | true  | false | false |
| SIX | true  | false | false | false | false |
| X   | false | false | false | false | false |



# Figure 15.17

| $T_{25}$            | $T_{26}$                                              |
|---------------------|-------------------------------------------------------|
| read ( $B$ )        | read ( $B$ )<br>$B := B - 50$<br>write ( $B$ )        |
| read ( $A$ )        | read ( $A$ )                                          |
| display ( $A + B$ ) | $A := A + 50$<br>write ( $A$ )<br>display ( $A + B$ ) |



# Figure 15.18

|               | $T_{27}$ | $T_{28}$      |
|---------------|----------|---------------|
| read ( $Q$ )  |          |               |
| write ( $Q$ ) |          | write ( $Q$ ) |



# Figure 15.19

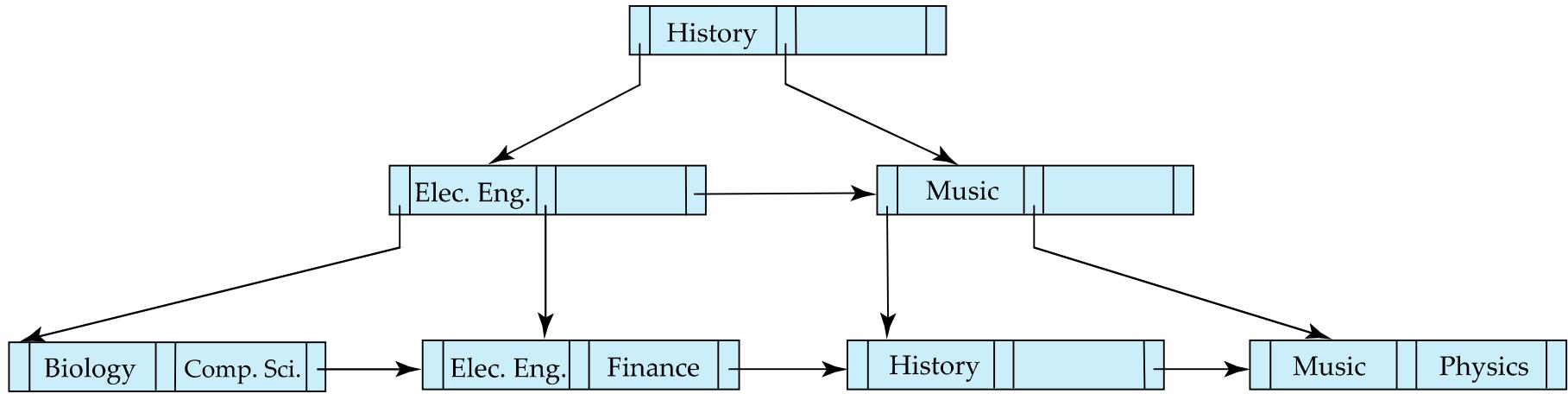
| $T_{25}$                                                          | $T_{26}$                                                       |
|-------------------------------------------------------------------|----------------------------------------------------------------|
| read ( $B$ )                                                      | read ( $B$ )<br>$B := B - 50$<br>read ( $A$ )<br>$A := A + 50$ |
| read ( $A$ )<br>$\langle validate \rangle$<br>display ( $A + B$ ) | $\langle validate \rangle$<br>write ( $B$ )<br>write ( $A$ )   |



# Figure 15.20

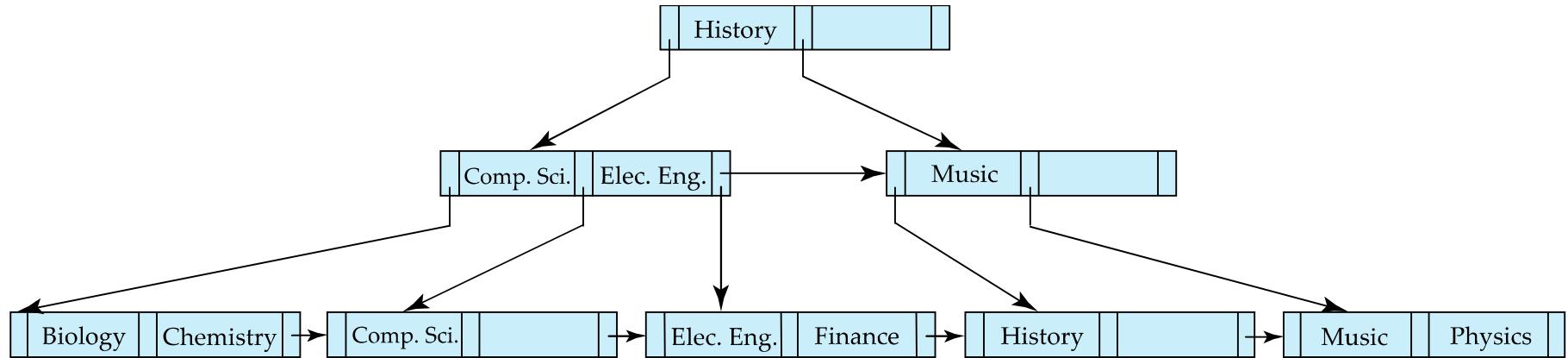
| $T_{32}$                                         | $T_{33}$                                                          |
|--------------------------------------------------|-------------------------------------------------------------------|
| lock-s ( $Q$ )<br>read ( $Q$ )<br>unlock ( $Q$ ) | lock-x ( $Q$ )<br>read ( $Q$ )<br>write ( $Q$ )<br>unlock ( $Q$ ) |
| lock-s ( $Q$ )<br>read ( $Q$ )<br>unlock ( $Q$ ) |                                                                   |

# Figure 15.21





# Figure 15.22





# Figure 15.23

|   | S     | X     | I     |
|---|-------|-------|-------|
| S | true  | false | false |
| X | false | false | false |
| I | false | false | true  |



# Figure in-15.1

| $T_{27}$      | $T_{28}$      | $T_{29}$      |
|---------------|---------------|---------------|
| read ( $Q$ )  | write ( $Q$ ) |               |
| write ( $Q$ ) |               | write ( $Q$ ) |

# Database design

Part 1

# Chapter 7: Entity-Relationship Model

- How is data stored in sql database - YouTube

# Chapter 7: Entity-Relationship Model

- Design Process
- Modeling
- Mapping Constraints
- E-R Diagram
- Design Issues
- Weak Entity Sets
- Extended E-R Features
- Design of the Bank Database
- Reduction to Relation Schemas
- Database Design
- UML



# DESIGN PROCESS

PPD

- Design Process
- E-R Model



## Design Phases

- The initial phase of database design is to characterize fully the data needs of the prospective database users
- Next, the designer chooses a data model and, by applying the concepts of the chosen data model, translates these requirements into a conceptual schema of the database
- A fully developed conceptual schema also indicates the functional requirements of the enterprise. In a “specification of functional requirements”, users describe the kinds of operations (or transactions) that will be performed on the data



## Design Phases (Cont.)

The process of moving from an abstract data model to the implementation of the database proceeds in two final design phases.

- 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 8)
    - Formalize what designs are bad, and test for them

PPD

- Design Process
- E-R Model



## E-R MODEL

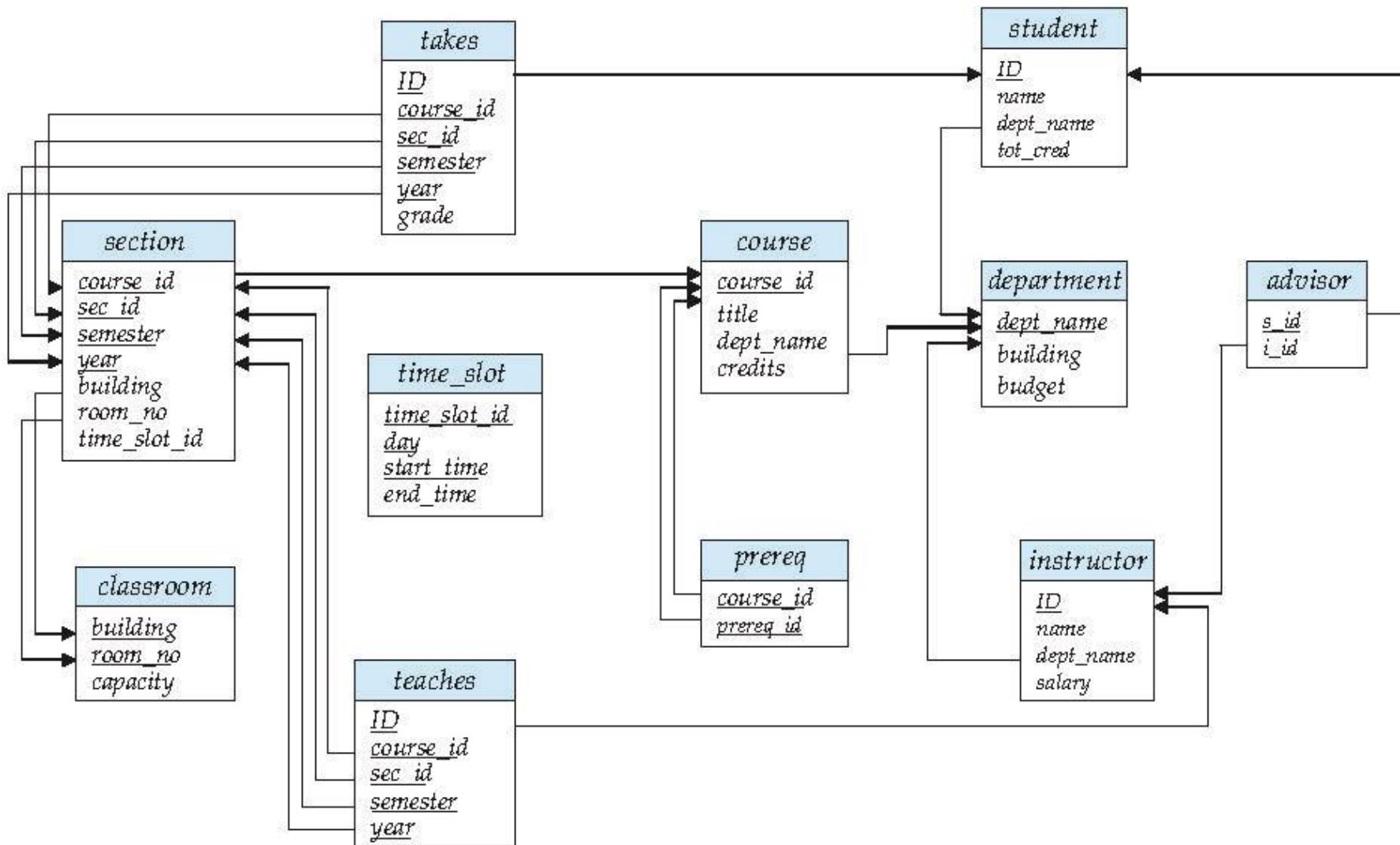


## ER model – Database Modeling

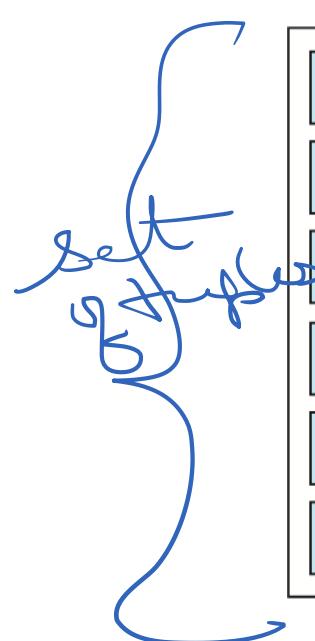
- The ER data model was developed to facilitate database design by allowing specification of an enterprise schema that represents the overall logical structure of a database
- The ER model is very useful in mapping the meanings and interactions of real-world enterprises onto a conceptual schema. Because of this usefulness, many database-design tools draw on concepts from the ER model
- The ER data model employs three basic concepts:
  - entity sets
  - relationship sets
  - attributes
- The ER model also has an associated diagrammatic representation, the ER diagram, which can express the overall logical structure of a database graphically

# Modeling

- A *database* can be modeled as:
  - a collection of entities,
  - relationship among entities.
- An **entity** is an object that exists and is distinguishable from other objects.
  - Example: person, company, event, plant, Course, Department
- Entities have **attributes**
  - Example: people have *names* and *addresses*
- 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



# Entity Sets *instructor* and *student*



instructor\_ID instructor\_name

|       |            |
|-------|------------|
| 76766 | Crick      |
| 45565 | Katz       |
| 10101 | Srinivasan |
| 98345 | Kim        |
| 76543 | Singh      |
| 22222 | Einstein   |

*instructor*

student-ID student\_name

|       |         |
|-------|---------|
| 98988 | Tanaka  |
| 12345 | Shankar |
| 00128 | Zhang   |
| 76543 | Brown   |
| 76653 | Aoi     |
| 23121 | Chavez  |
| 44553 | Peltier |

*student*

# Relationship Sets

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

Example:

44553 (Austin)      *advisor*      22222 (Einstein)  
~~student entity~~ relationship set ~~instructor~~ entity

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

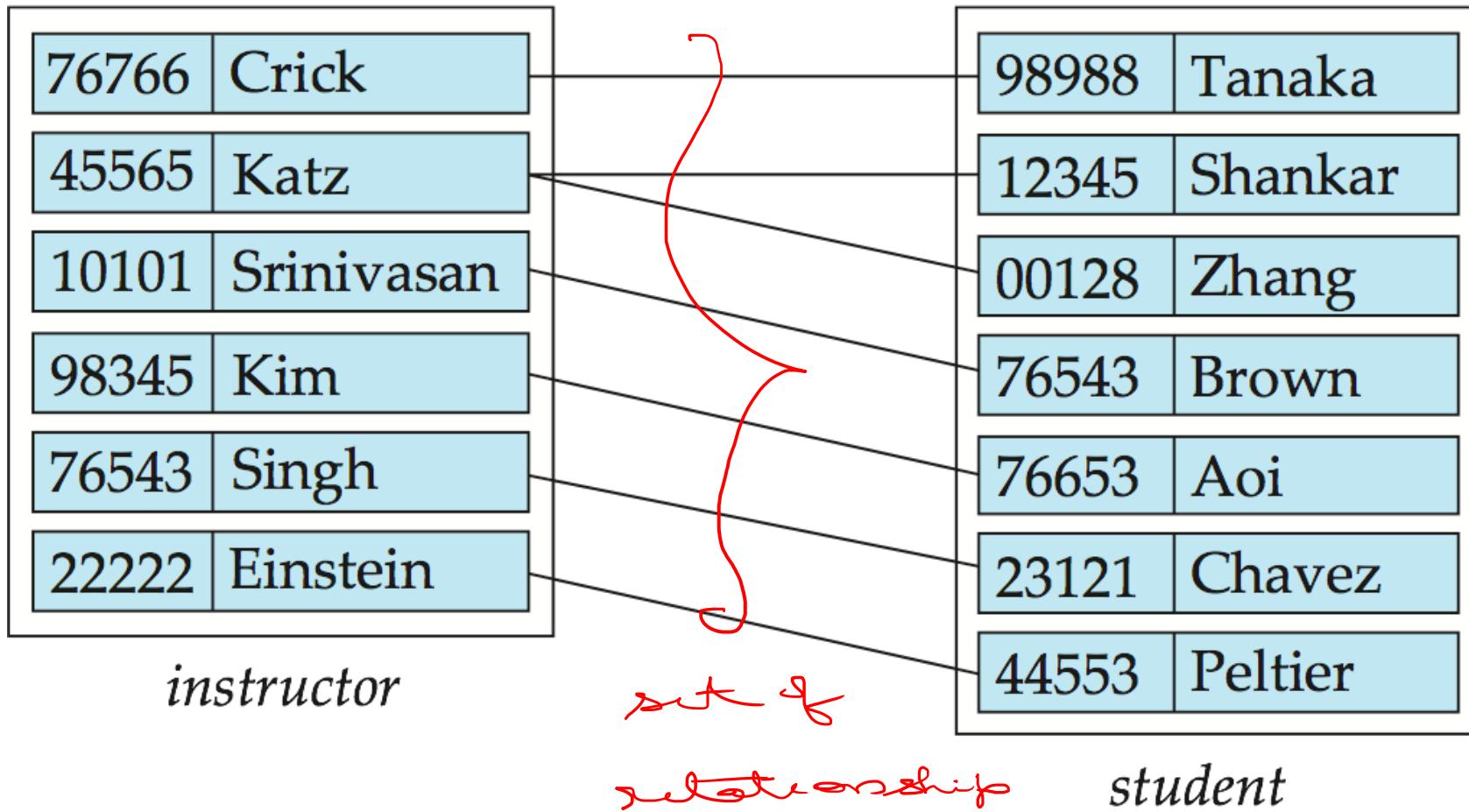
$$\{(e_1, e_2, \dots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \dots, e_n \in E_n\}$$

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

- Example:

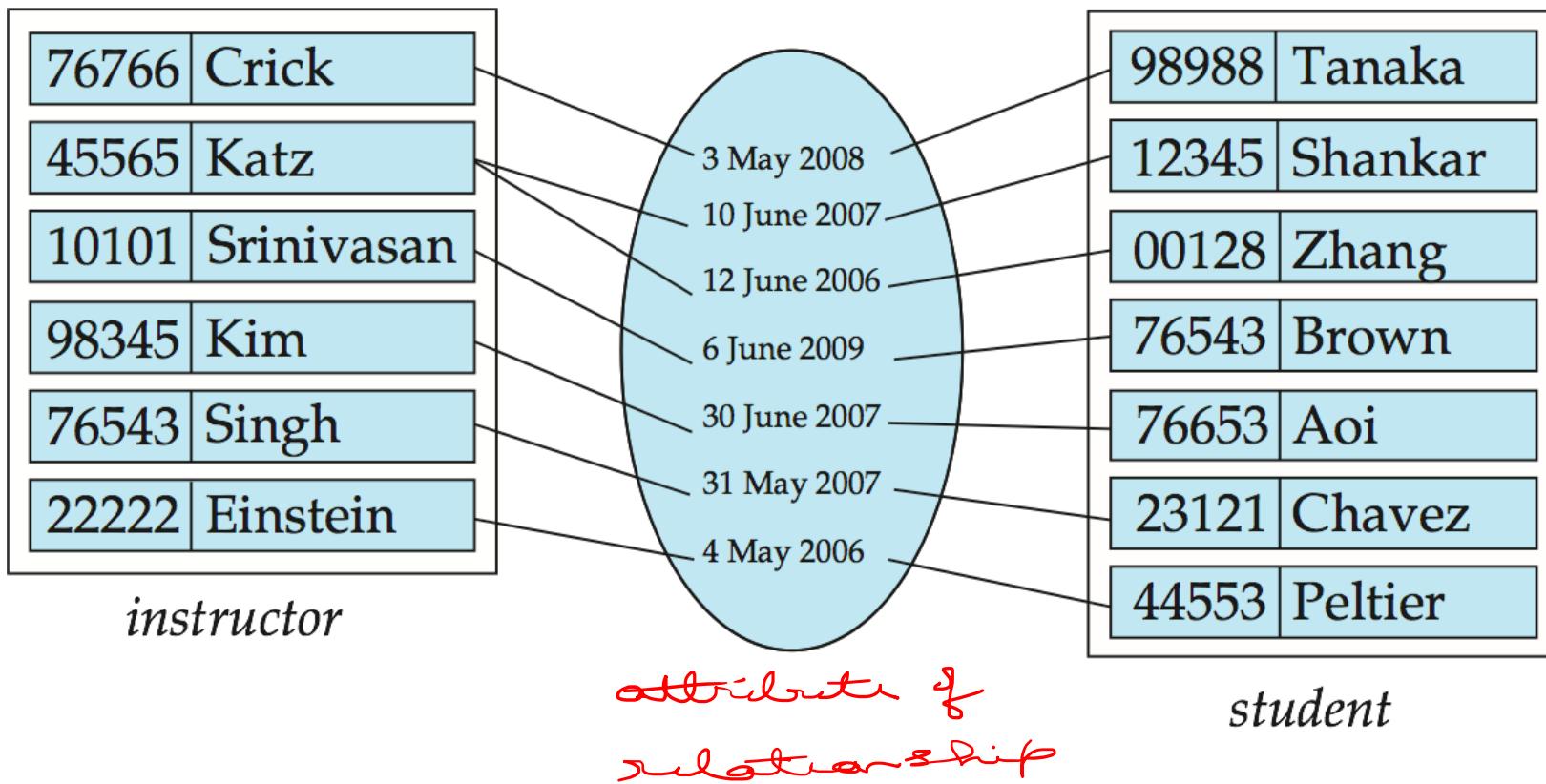
$$(44553, 22222) \in \text{advisor}$$

# Relationship Set advisor



# Relationship Sets (Cont.)

- A **descriptive attribute** can also be **property of 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



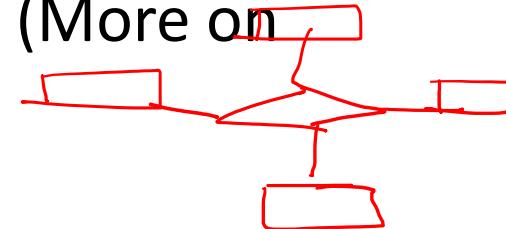
# Degree of a Relationship Set

- **binary relationship**

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



- 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.

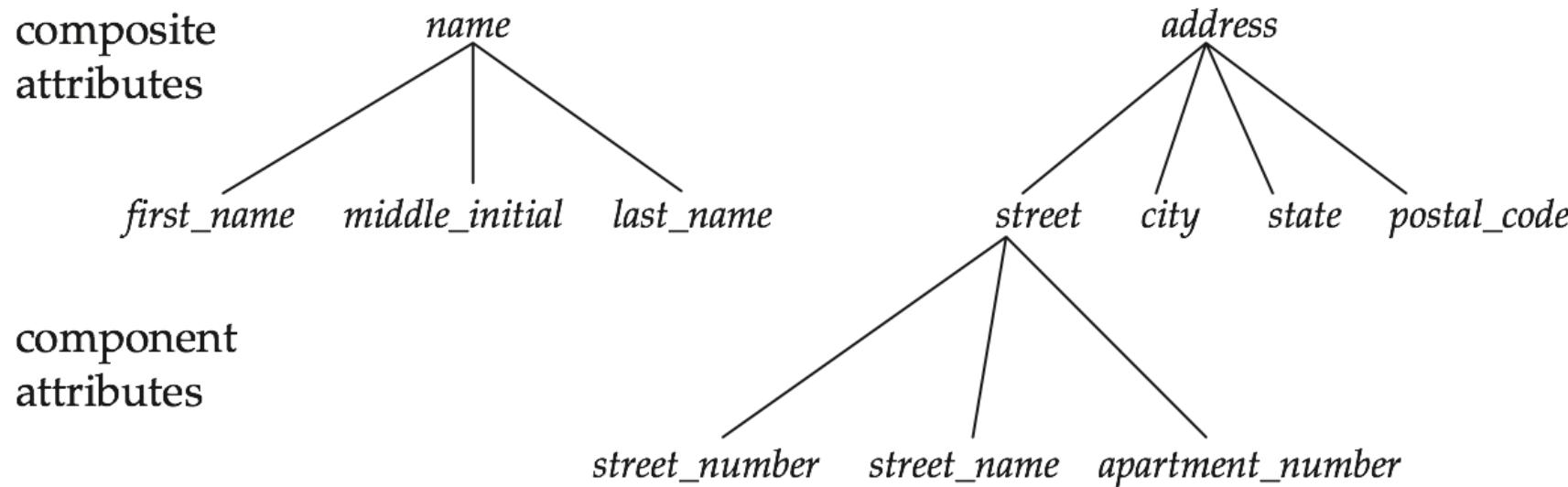
- Example:

*instructor = (ID, name, street, city, salary )  
course= (course\_id, title, credits)*

- **Domain** – the set of permitted values for each attribute
- **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

*Ex :*

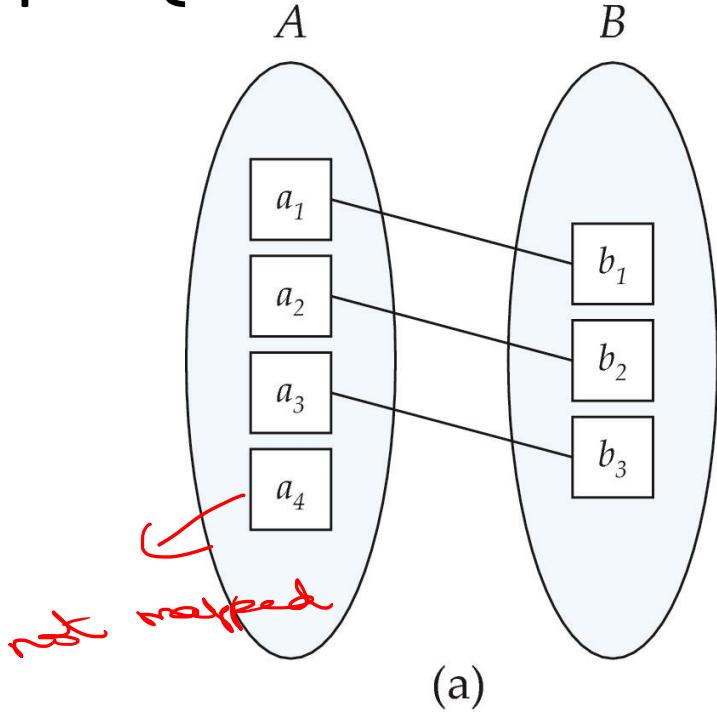
# Composite Attributes



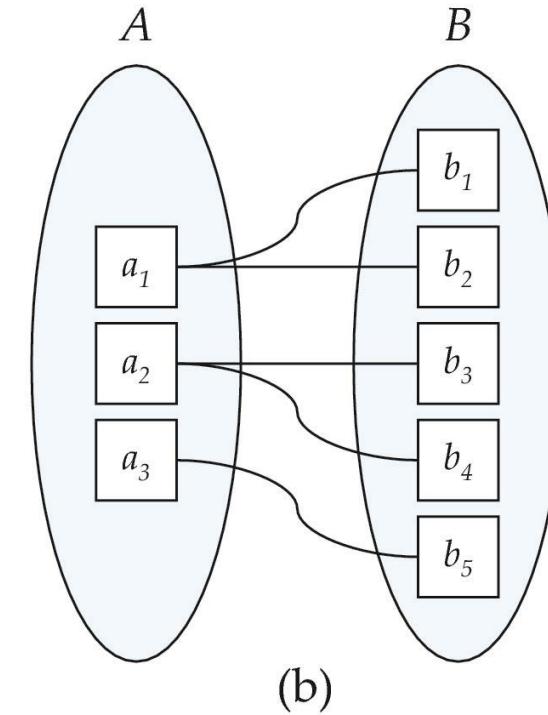
# Mapping Cardinality Constraints

- 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



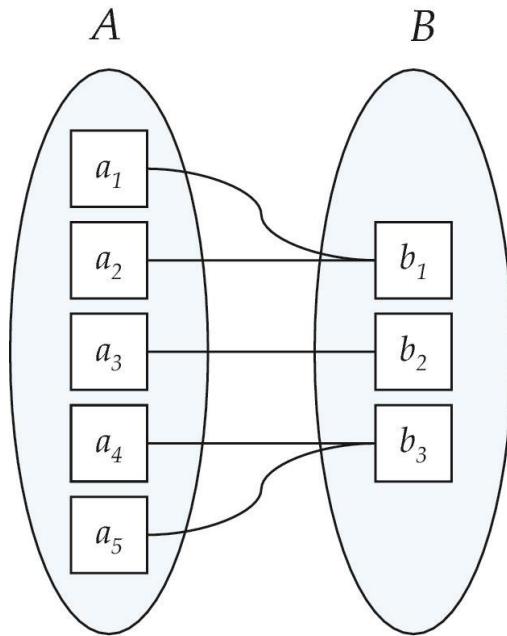
One to one



One to many

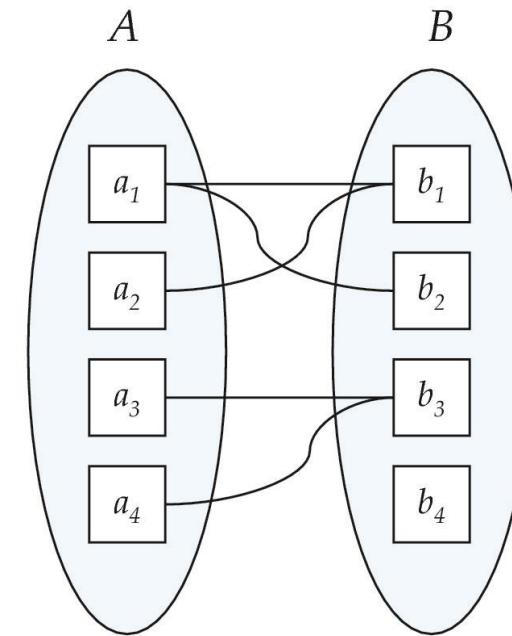
Note: Some elements in  $A$  and  $B$  may not be mapped to any elements in the other set

# 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

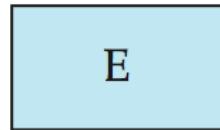
# ER DIAGRAM

## CHEN'S NOTATION

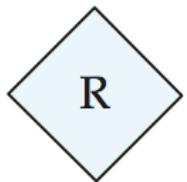
---

To be preferably followed

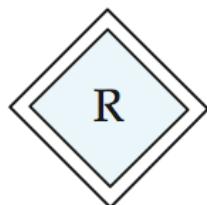
# Symbols Used in E-R Notation



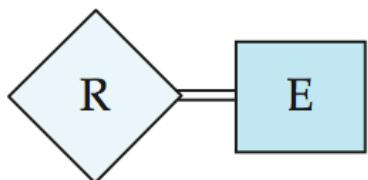
entity set



relationship set



identifying  
relationship set  
for weak entity set



total participation  
of entity set in  
relationship

|      |
|------|
| E    |
| A1   |
| A2   |
| A2.1 |
| A2.2 |
| {A3} |
| A4() |

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

|           |
|-----------|
| E         |
| <u>A1</u> |

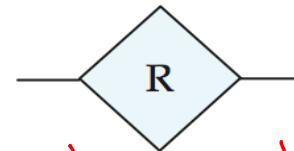
primary key

|       |
|-------|
| E     |
| A1    |
| ..... |

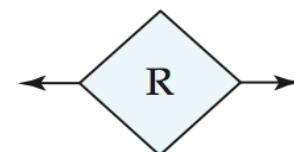
discriminating  
attribute of  
weak entity set

# Symbols Used in E-R Notation (Cont.)

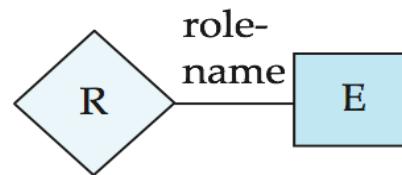
~~Relationship symbols~~



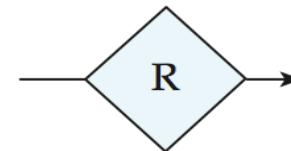
many-to-many  
relationship



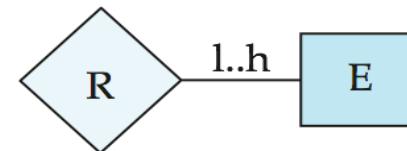
one-to-one  
relationship



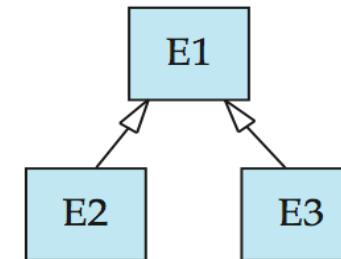
role indicator



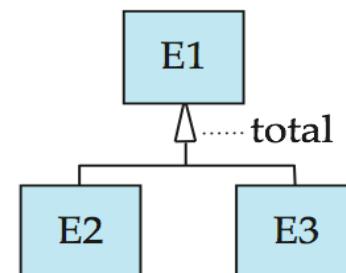
many-to-one  
relationship



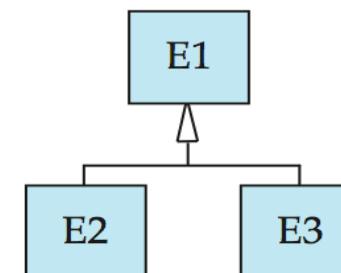
cardinality  
limits



ISA: generalization  
or specialization



total (disjoint)  
generalization

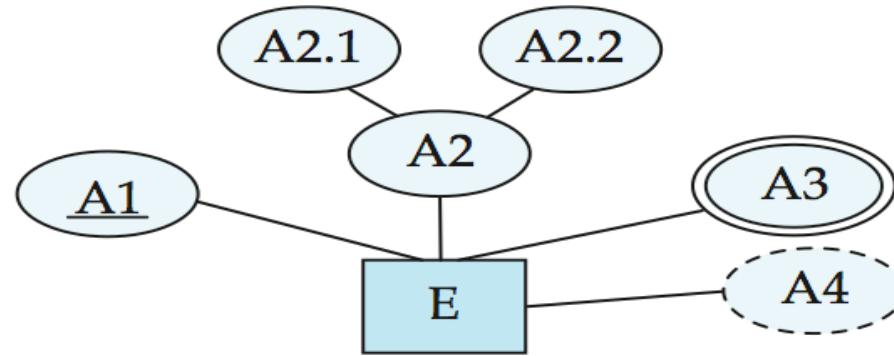


disjoint  
generalization

# Alternative ER Notations

- Chen, IDE1FX, ...

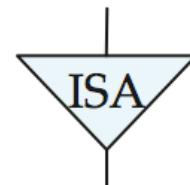
entity set E with  
simple attribute A1,  
composite attribute A2,  
multivalued attribute A3,  
derived attribute A4,  
and primary key A1



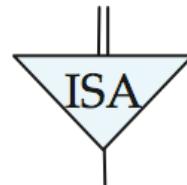
weak entity set



generalization



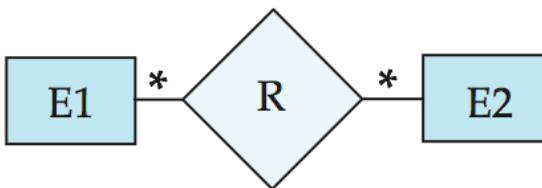
total  
generalization



# Alternative ER Notations

Chen

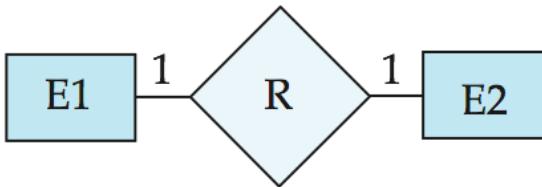
many-to-many  
relationship



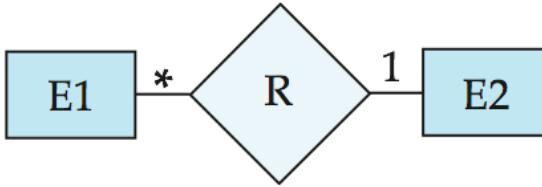
IDE1FX (**Crows feet notation**)

~~IDE1FX~~

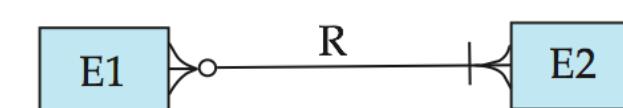
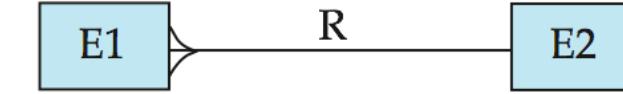
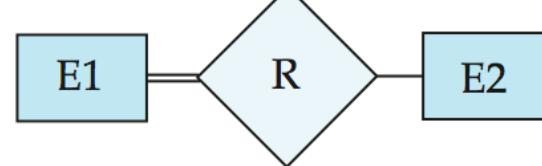
one-to-one  
relationship



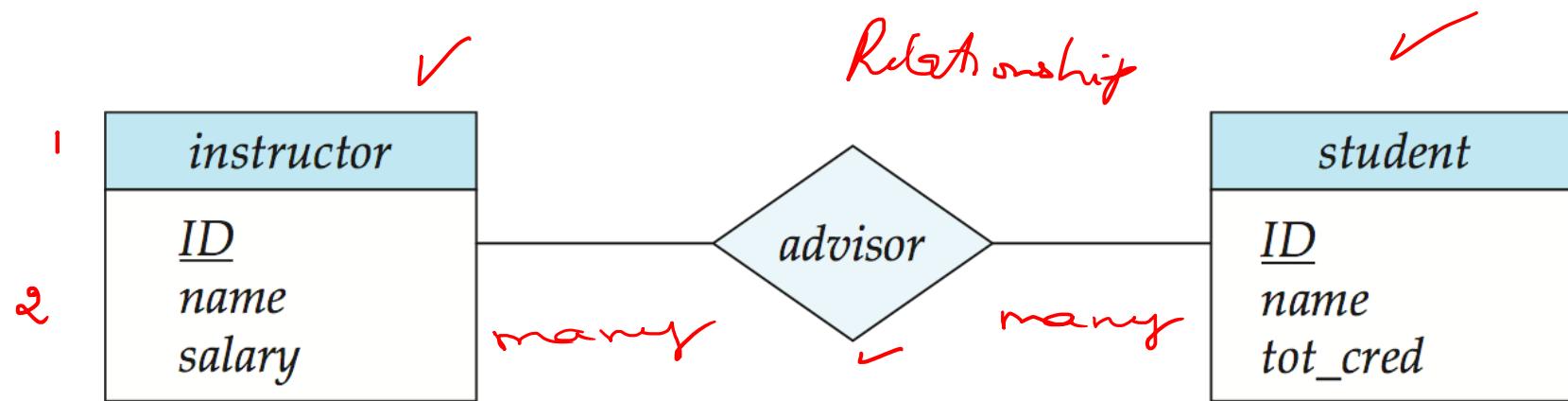
many-to-one  
relationship



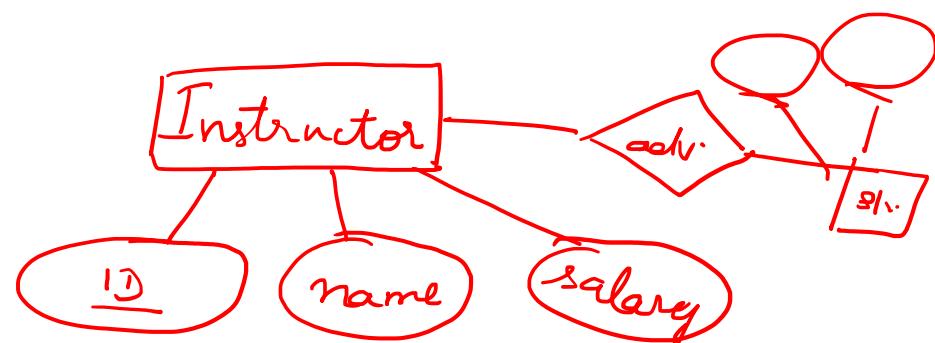
participation  
in R: total (E1)  
and partial (E2)



# E-R Diagrams



- n Rectangles represent entity sets.
- n Diamonds represent relationship sets.
- n Attributes listed inside entity rectangle
- n Underline indicates primary key attributes



# Entity With Composite, Multivalued, and Derived Attributes

phone-number

: - age - ;

ER component

| instructor |                                             |
|------------|---------------------------------------------|
| 1          | <u>ID</u>                                   |
| 2          | <u>name</u>                                 |
|            | ↳ first_name<br>middle_initial<br>last_name |
| 3          | <u>address</u>                              |
|            | street                                      |
|            | street_number                               |
|            | street_name                                 |
|            | apt_number }                                |
|            | city                                        |
|            | state                                       |
|            | zip                                         |
| 4          | { phone_number }                            |
| 5          | <u>date_of_birth</u>                        |
| 6          | <u>age ()</u>                               |

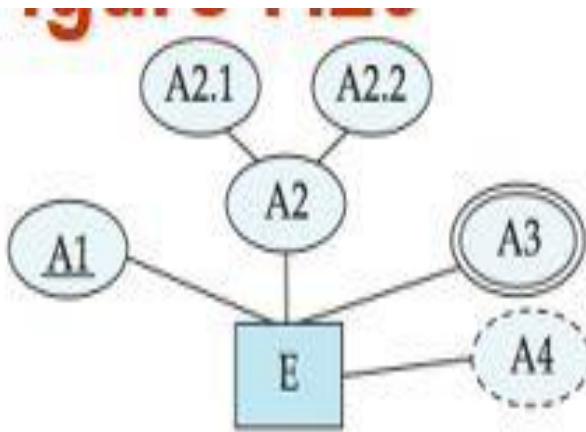
Alternative form



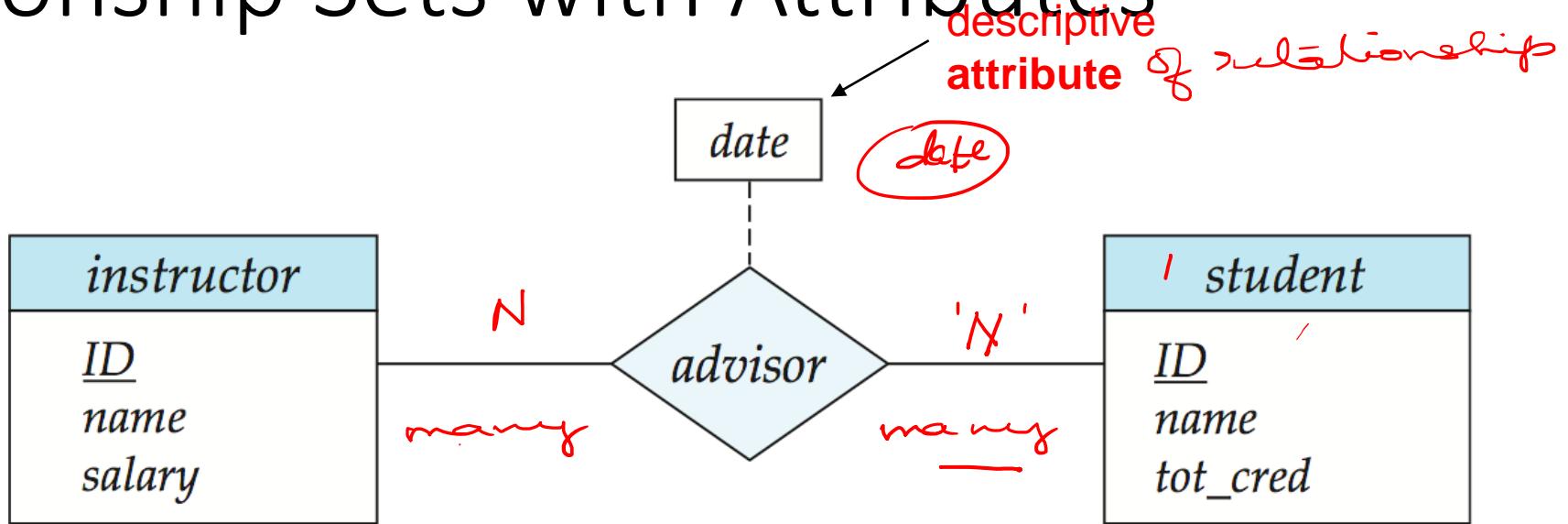
Note :-

Preferable representation

entity set E with  
simple attribute A1,  
composite attribute A2,  
multivalued attribute A3,  
derived attribute A4,  
and primary key A1



# Relationship Sets with Attributes

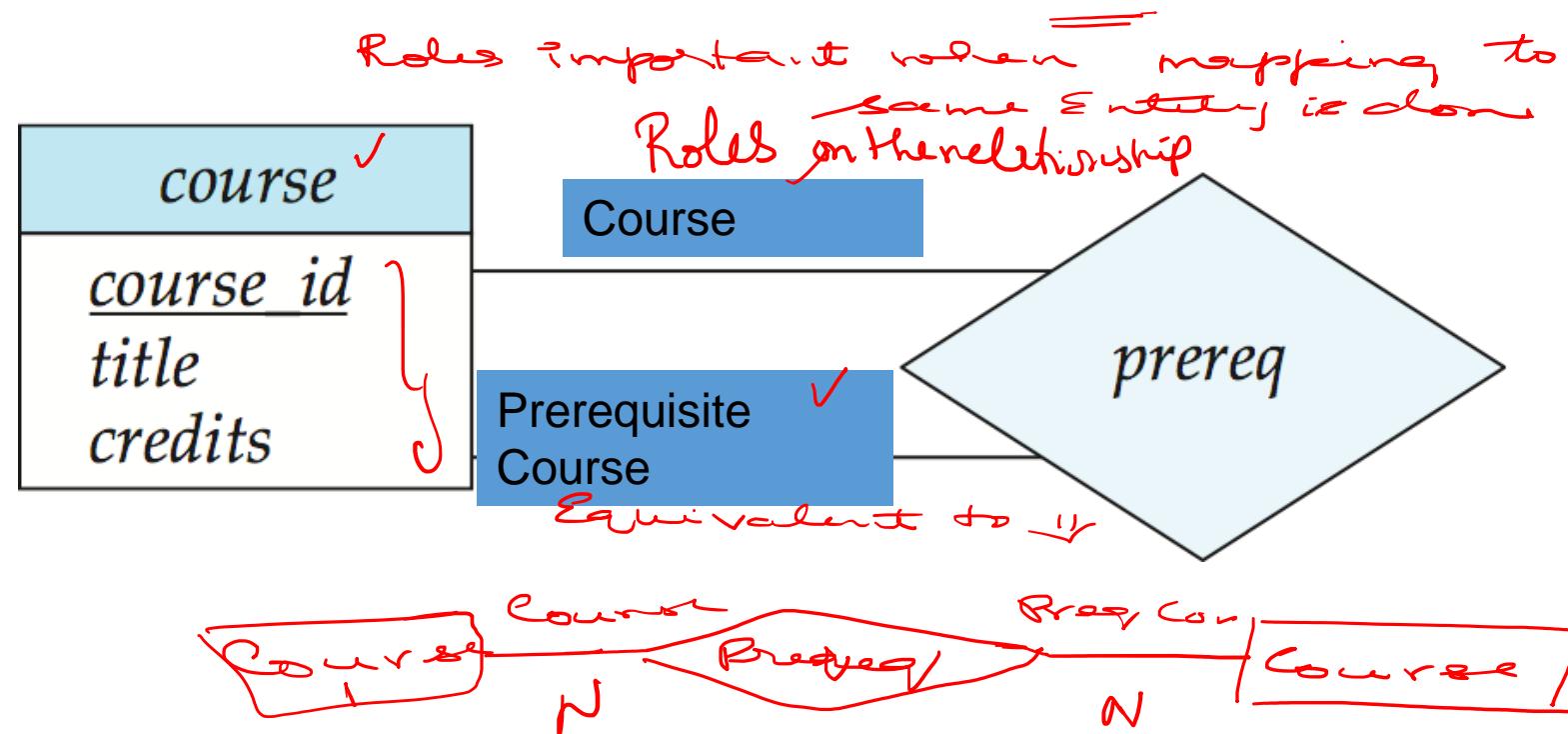


→ One how many instances are associated  
— Many

# Roles

one of the Employee is head of the dept

- 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**.



constraints → relationship

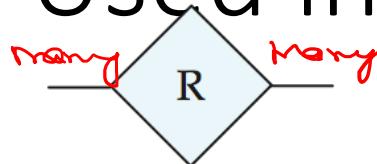
- Cardinality
- participation

# Cardinality Constraints

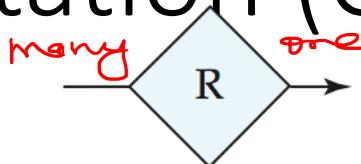
- We express cardinality constraints by drawing either a directed line ( $\rightarrow$ ), signifying “one,” or an undirected line ( $-$ ), signifying “many” between the relationship set and the entity set.

*Note: many includes  $\rightarrow$*

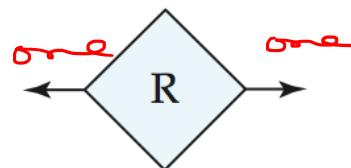
# Symbols Used in E-R Notation (Cont.)



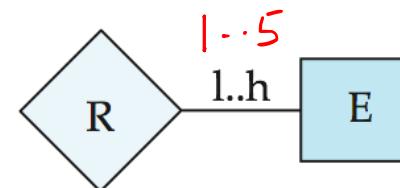
many-to-many  
relationship



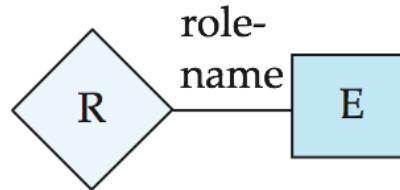
many-to-one  
relationship



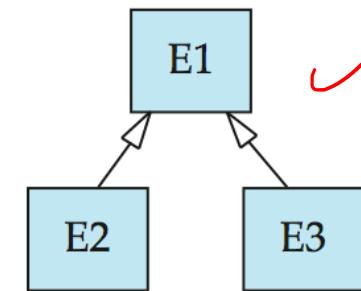
one-to-one  
relationship



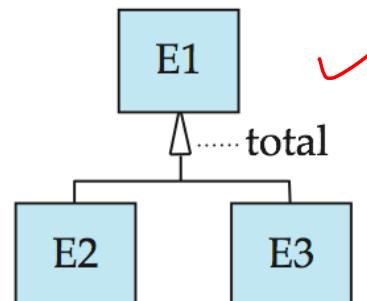
cardinality  
limits



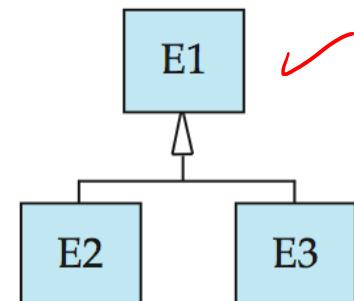
role indicator



ISA: generalization  
or specialization



total (disjoint)  
generalization

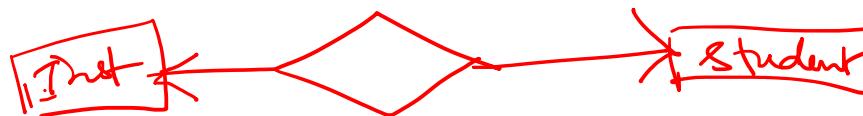
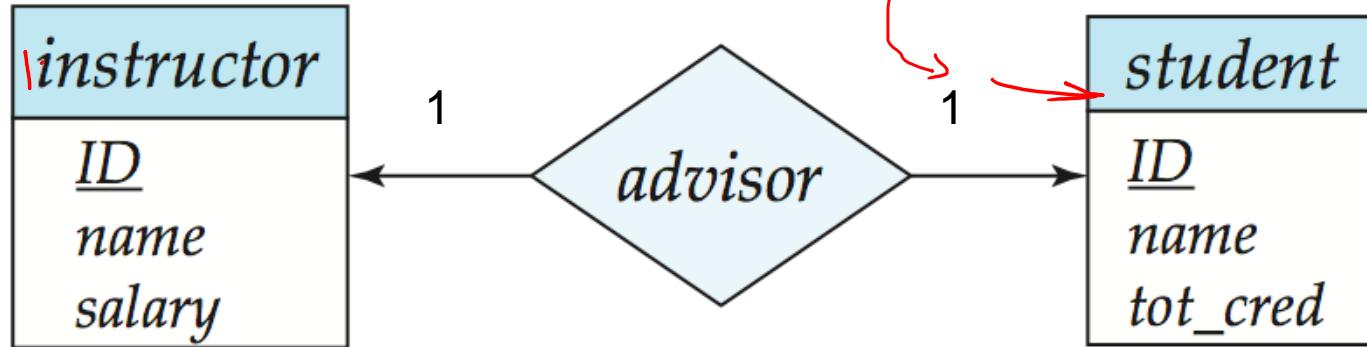


disjoint  
generalization

# One-to-One Relationship

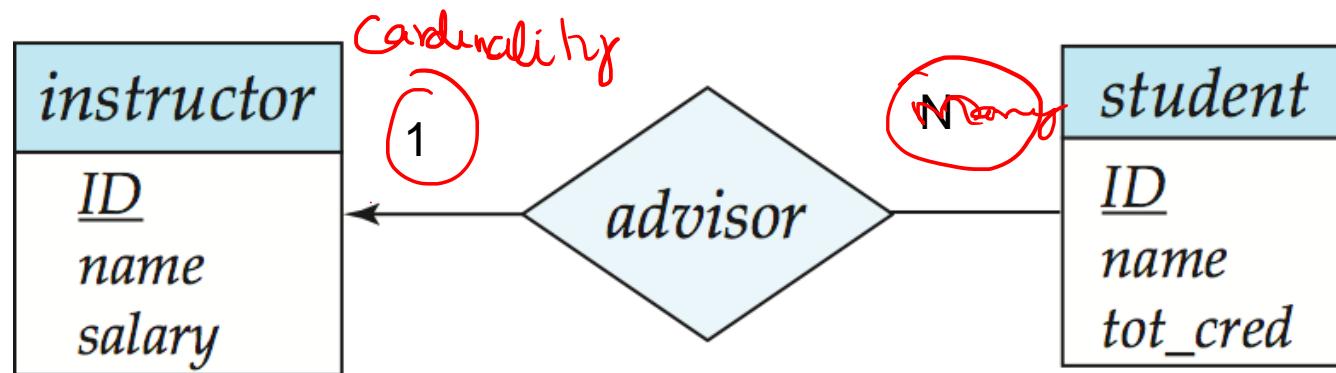
- one-to-one relationship between an *instructor* and a *student*
  - an instructor is associated with **at most one** student via *advisor*
  - and a student is associated with **at most one** instructor via *advisor*

*each instructor associates with*



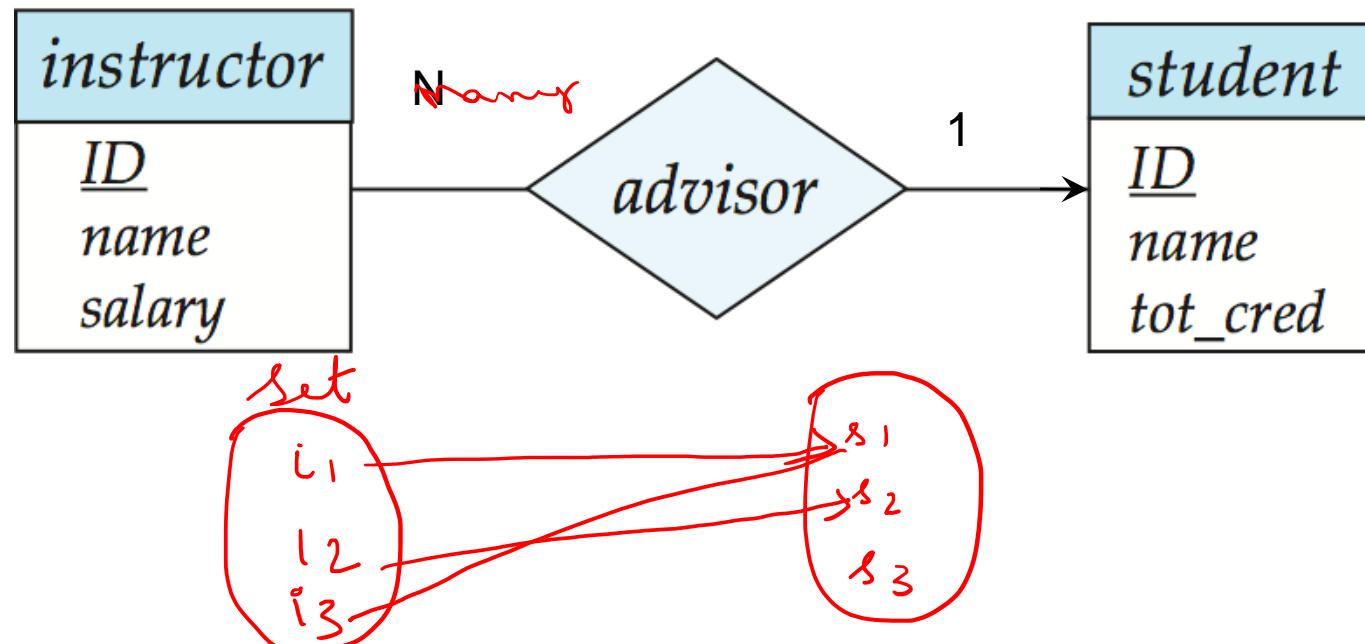
# One-to-Many Relationship

- 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,



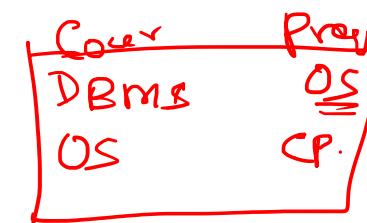
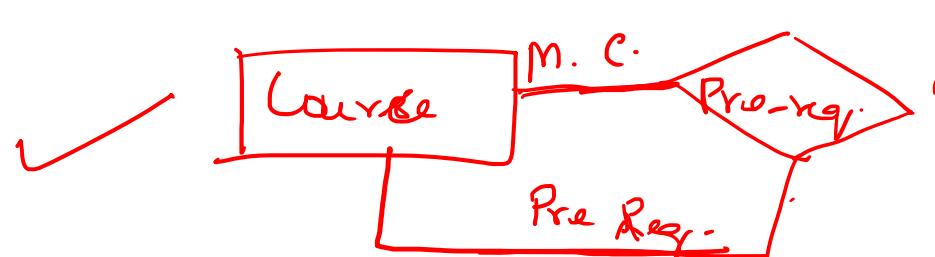
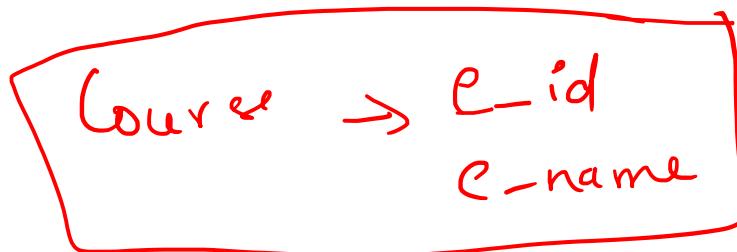
# Many-to-One Relationships

- 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*



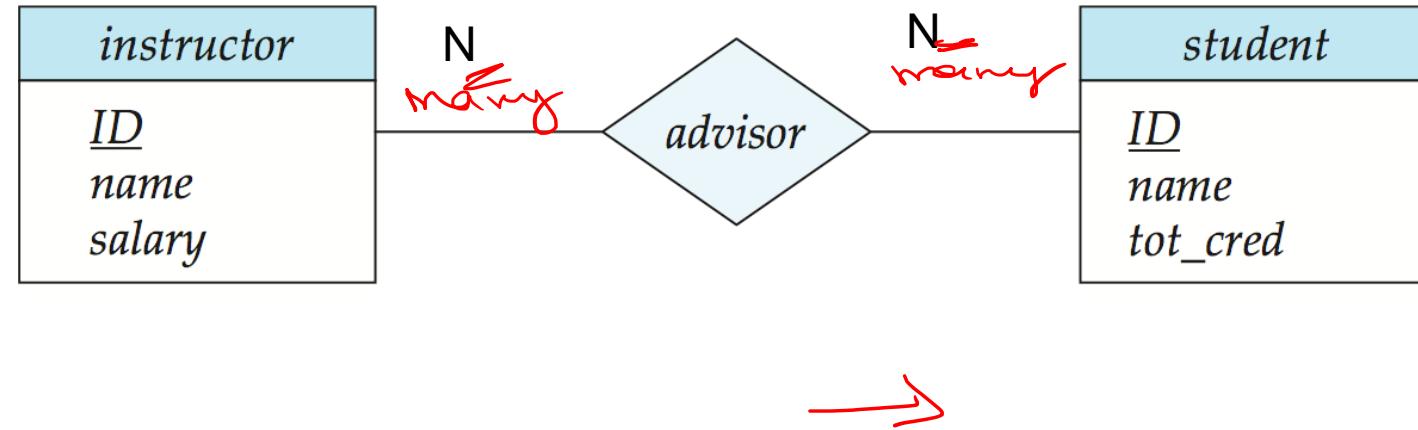
Roles :- same entity is mapped to itself

Course has pre requisite Course.



# 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*



Ports  
 $P_1$   
 $P_2$   
 $P_3 \checkmark$

model  
 $m_1$   
 $m_2$

$m_1 = P_1 + P_2$   
 $m_2 = P_2$

## Participation of an Entity Set in a Relationship Set

n Total participation (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set
   
   | E.g., participation of *section* in *sec\_course* is total
   
     ▶ *every section must have an associated course*

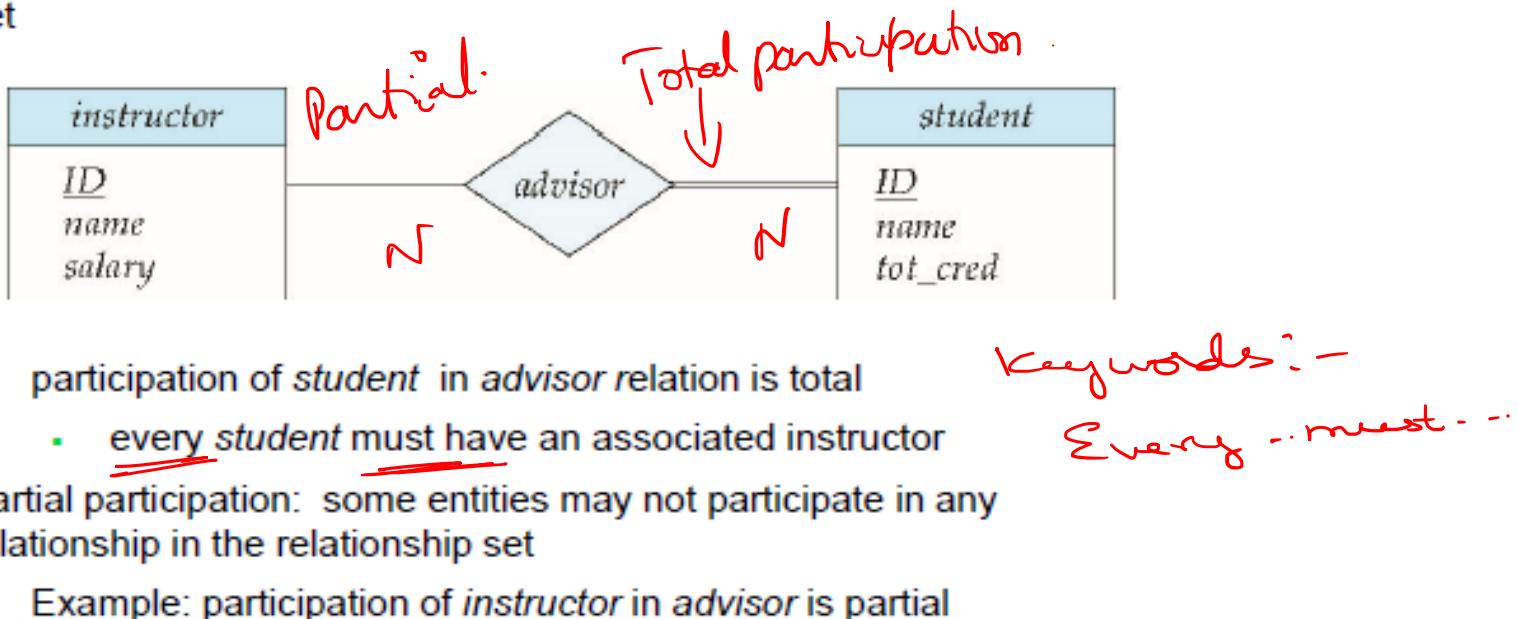
n Partial participation: some entities may not participate in any relationship in the relationship set
   
   | Example: participation of *instructor* in *advisor* is partial
   
     *if there are no sections then all are mapped*

Total participation

Total with '1'  
 Total with N

## Total and Partial Participation

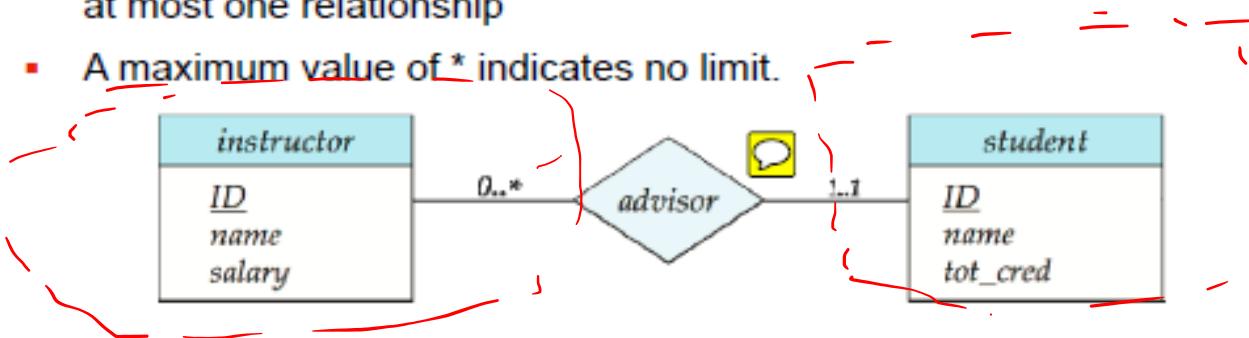
- Total participation (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set



## 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.

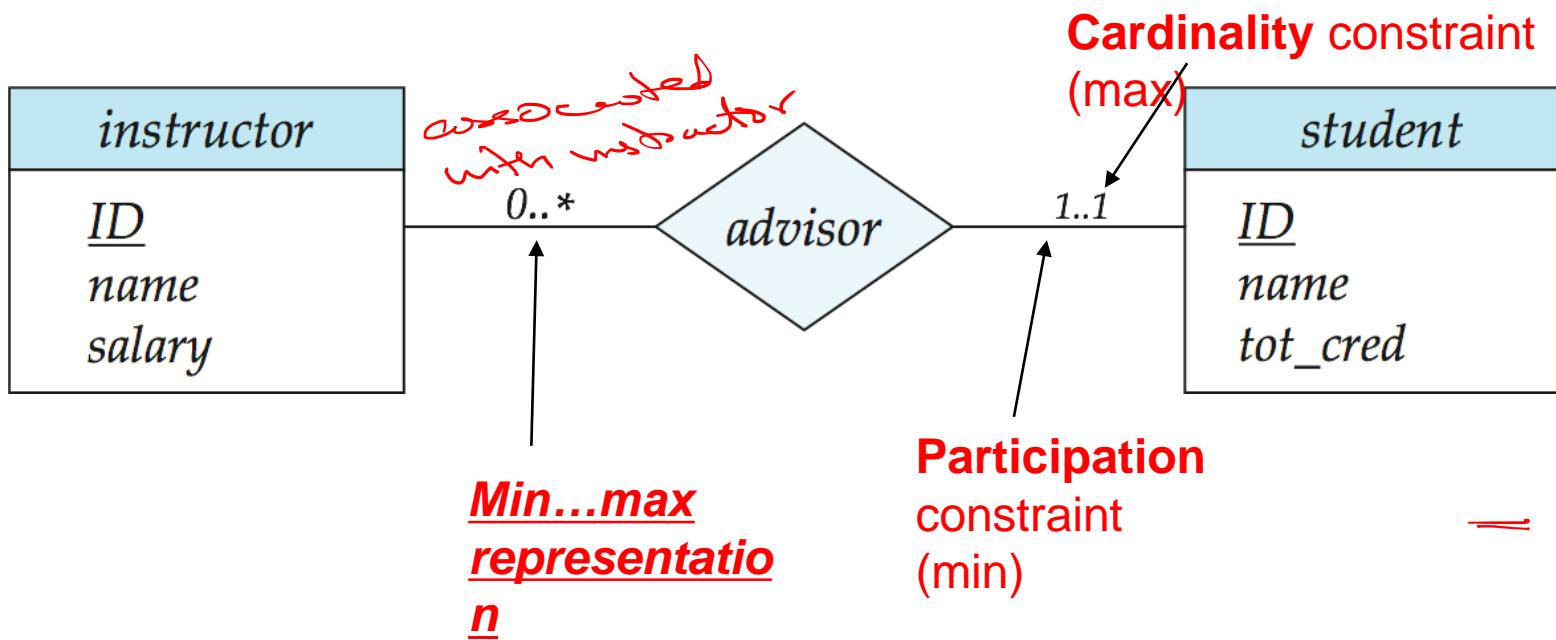


Instructor can advise 0 or more students. A student must have 1 advisor; cannot have multiple advisors

Note: cardinality associated with the entity close to where it is written

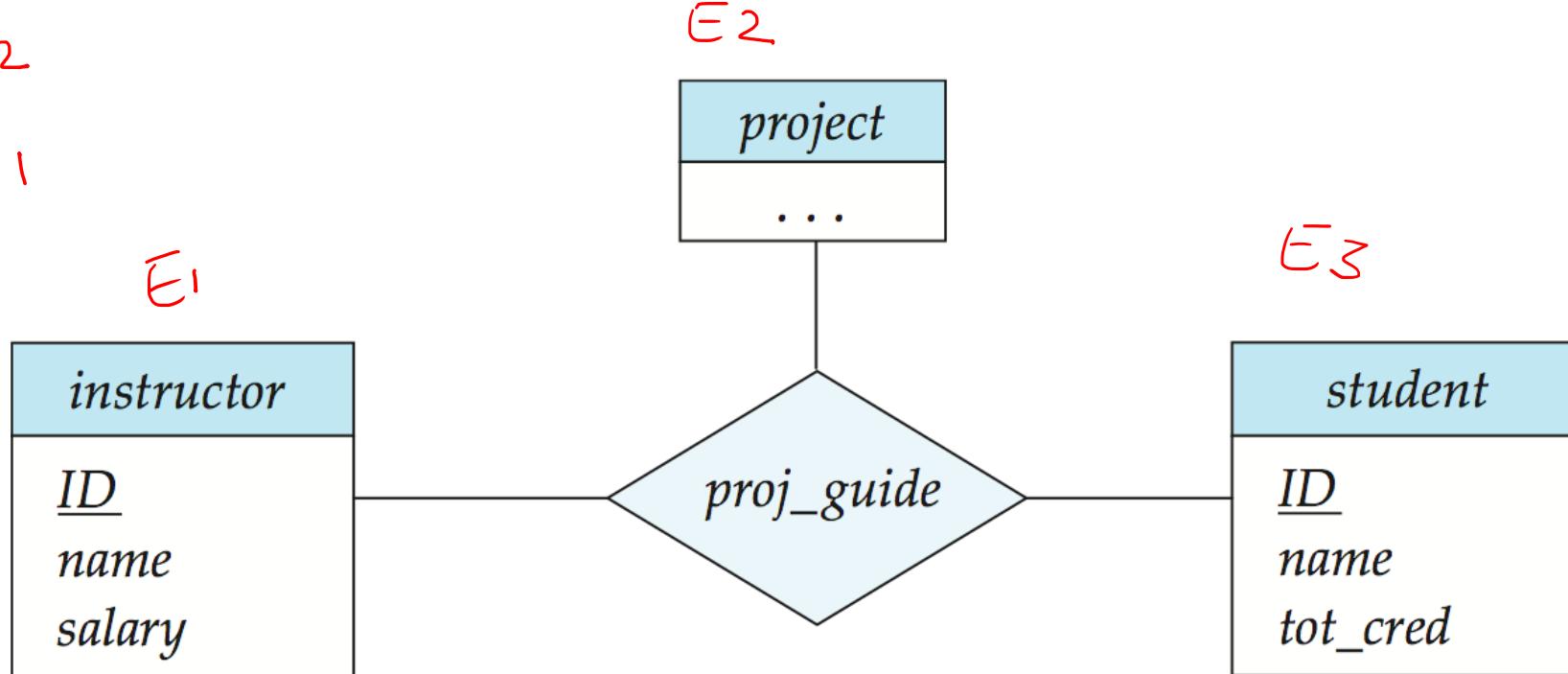
# Alternative Notation for Cardinality Limits

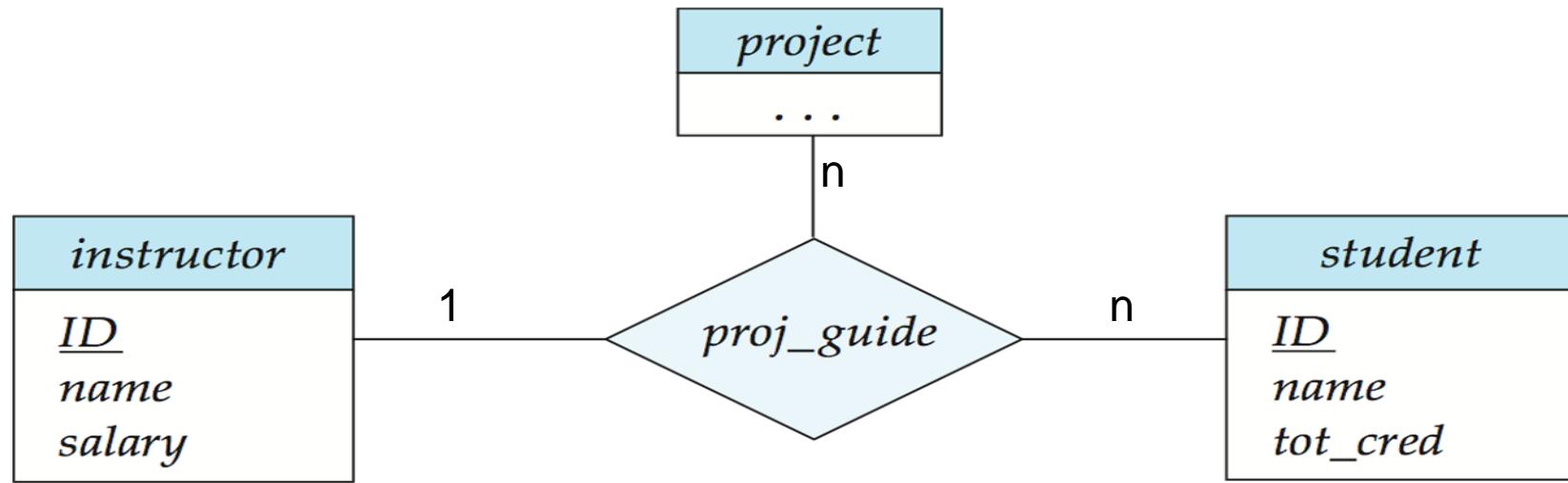
- Cardinality limits can also express participation constraints



# E-R Diagram with a Ternary Relationship

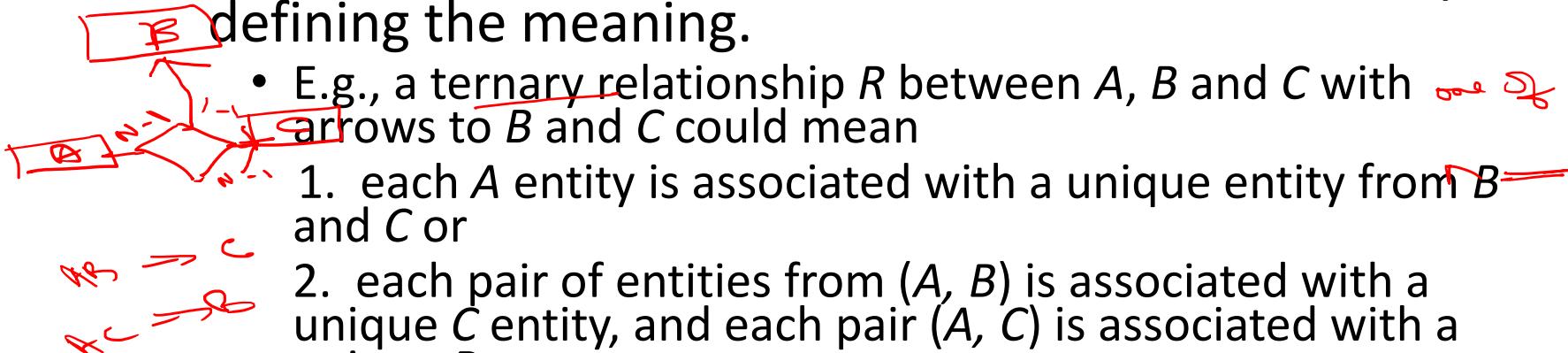
I<sub>1</sub> P<sub>1</sub> S<sub>1</sub>  
I<sub>1</sub> P<sub>1</sub> S<sub>2</sub>  
I<sub>2</sub> P<sub>2</sub> S<sub>1</sub>  
I<sub>2</sub> P<sub>1</sub> S<sub>1</sub>





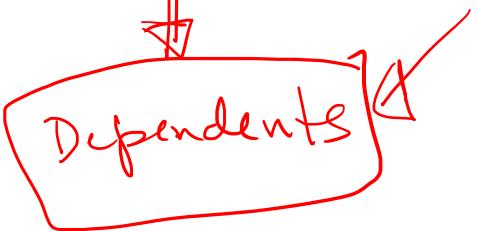
# Cardinality Constraints on Ternary Relationship

- We allow at most one arrow out of a ternary (or greater degree) relationship to indicate a cardinality constraint
- E.g., an arrow from *proj\_guide* to *instructor* indicates each student has at most one guide for a project
- If there is more than one arrow, there are two ways of defining the meaning.

- 
- E.g., a ternary relationship  $R$  between  $A$ ,  $B$  and  $C$  with ~~one~~ ~~two~~ arrows to  $B$  and  $C$  could mean
    1. each  $A$  entity is associated with a unique entity from  $B$  and  $C$  or
    2. each pair of entities from  $(A, B)$  is associated with a unique  $C$  entity, and each pair  $(A, C)$  is associated with a unique  $B$
  - Each alternative has been used in different formalisms
  - To avoid confusion we outlaw more than one arrow

# Weak Entity Sets

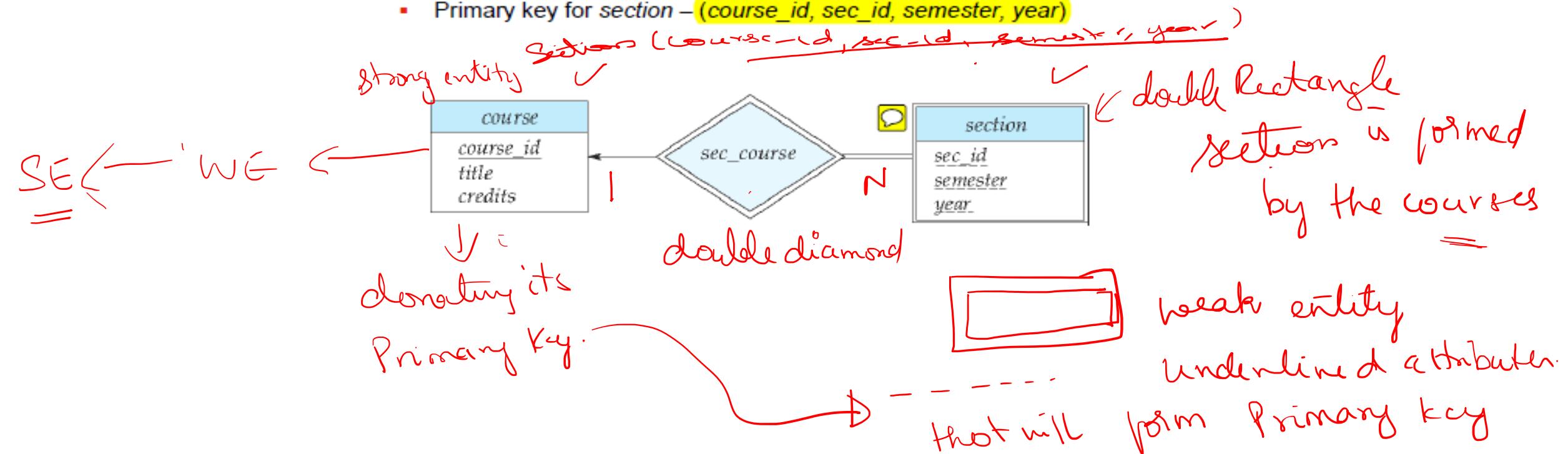
- An entity set that does not have a primary key is referred to as a **weak entity set**.
- The existence of a weak entity set depends on the existence of a **identifying entity set**
  - It must relate to the identifying entity set via a total, one-to-many relationship set from the identifying to the weak entity set
  - **Identifying relationship** depicted using a double diamond
- The **discriminator (or partial key) of a weak entity set** is the set of attributes that distinguishes among all the entities of a weak entity set.
- The **primary key of a weak entity** set is formed by the primary key of the strong entity set on which the weak entity set is existence dependent, plus the weak entity set's discriminator.



Identifying relationship

## Expressing Weak Entity Sets

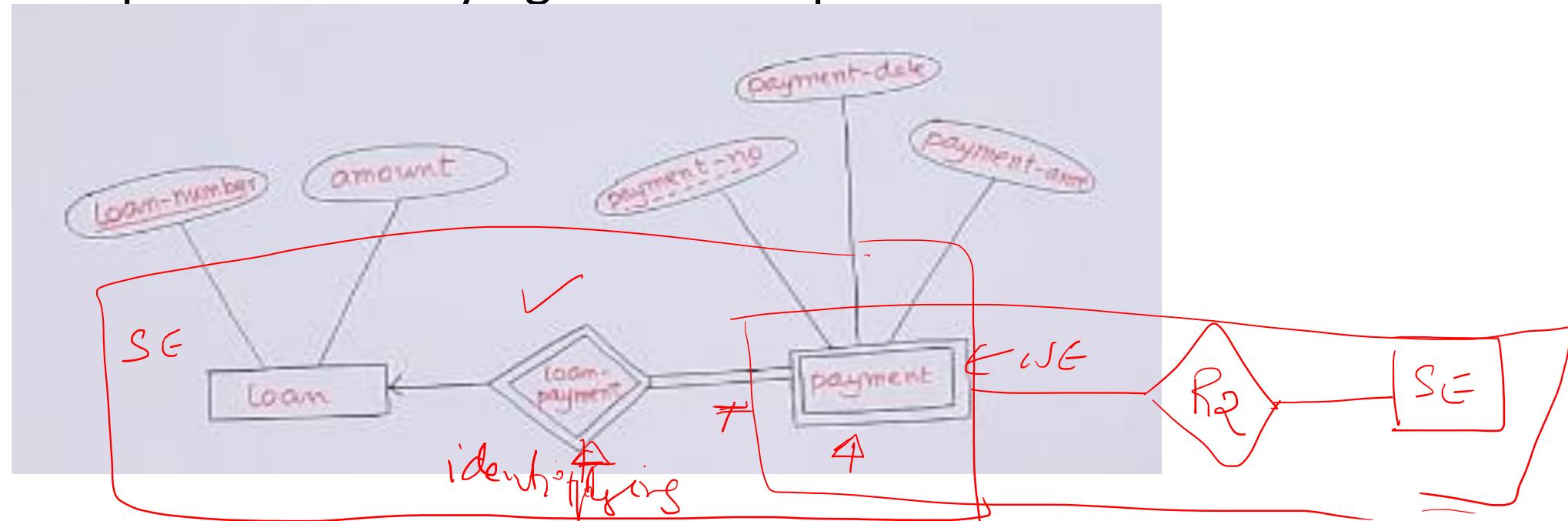
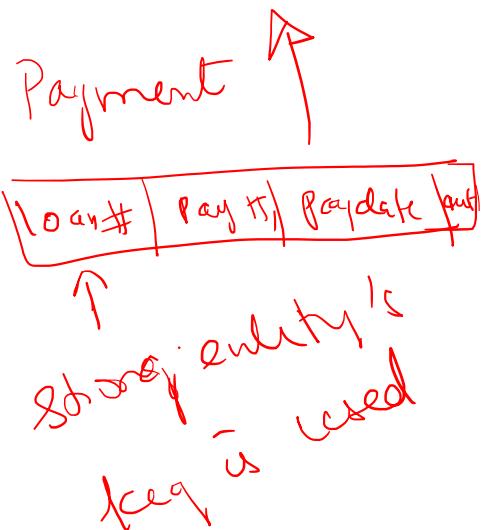
- In E-R diagrams, a weak entity set is depicted via a double rectangle
- We underline the discriminator of a weak entity set with a dashed line
- The relationship set connecting the weak entity set to the identifying strong entity set is depicted by a double diamond
- Primary key for section – (course\_id,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



# Weak Entity Sets (Cont.)

|   |     |      |      |   |
|---|-----|------|------|---|
| 1 | 111 | 21-5 | 1000 | } |
| 2 | 111 | 21-5 | 1000 |   |

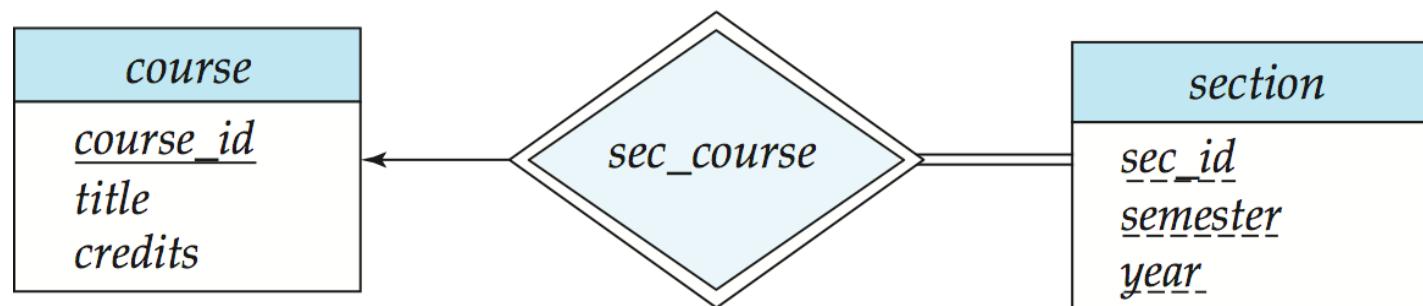
- We underline the **discriminator (or partial key)** of a weak entity set with a dashed line.
- We put the identifying relationship of a weak



The **primary key** of a weak entity set is formed by the **primary key** of the strong entity set on which the weak entity set is existence dependent, **plus** the weak entity set's discriminator.

# Weak Entity Sets (Cont.)

- Note: the primary key of the strong entity set is not explicitly stored with the weak entity set, since it is implicit in the identifying relationship.
- If *course\_id* were explicitly stored, *section* could be made a strong entity, but then the relationship between *section* and *course* would be duplicated by an implicit relationship defined by the attribute *course\_id* common to *course* and *section*
- Primary key for *section* – (*course\_id*, *sec\_id*, *semester*, *year*)



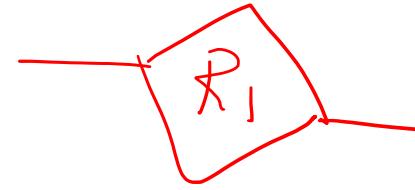
# Removing redundant attributes in entity sets

- Once the entities and their corresponding attributes are chosen, the relationship sets among the various entities are formed. These relationship sets may result in a situation where attributes in the various entity sets are redundant and need to be removed from the original entity sets.

- the attribute *dept name* in fact gets added to the relation *instructor*, but only if each instructor has at most one associated department.
- If an instructor has more than one associated department, the relationship between instructors and departments is recorded in a separate relation *inst dept*.
- Ex2: sec-timeslot
- And so on

For our university example, we list the entity sets and their attributes below, with primary keys underlined:

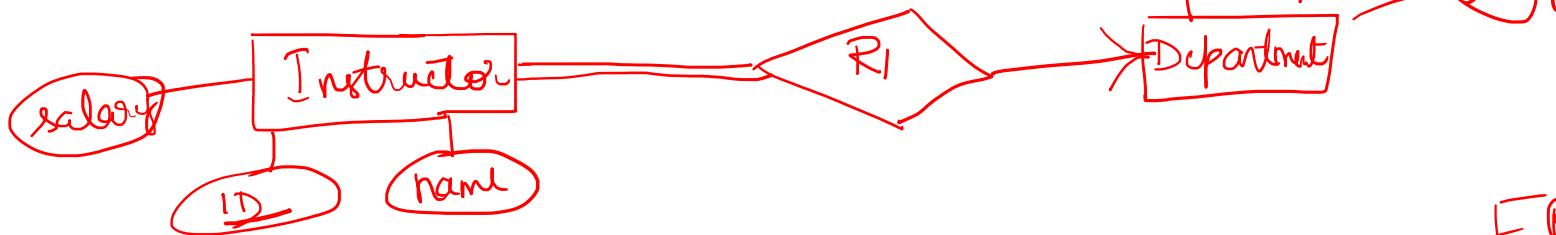
1. **classroom**: with attributes (*building, room number, capacity*).
2. **department**: with attributes (*dept name, building, budget*).
3. **course**: with attributes (*course id, title, credits*).
4. **instructor**: with attributes (*ID, name, salary*).
5. **section**: with attributes (*course id, sec id, semester, year*).
6. **student**: with attributes (*ID, name, tot cred*).
7. **time slot**: with attributes (*time slot id, {(day, start time, end time)}*).



The relationship sets in our design are listed below:

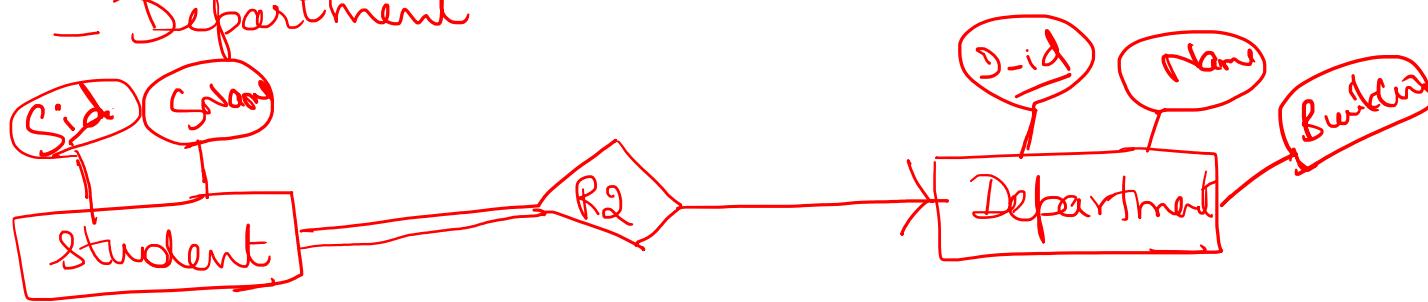
- **inst dept**: relating instructors with departments.
- **stud dept**: relating students with departments.
- **teaches**: relating instructors with sections.
- **takes**: relating students with sections, with a descriptive attribute *grade*.
- **course dept**: relating courses with departments.
- **sec course**: relating sections with courses.
- **sec class**: relating sections with classrooms.
- **sec time slot**: relating sections with time slots.
- **advisor**: relating students with instructors.
- **prereq**: relating courses with prerequisite courses.

① Instructor - dept

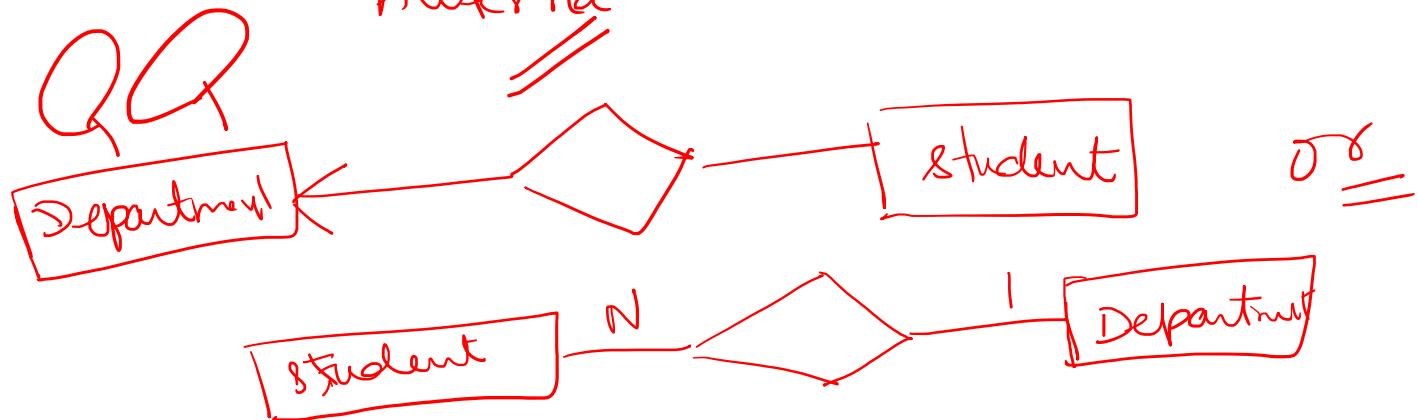


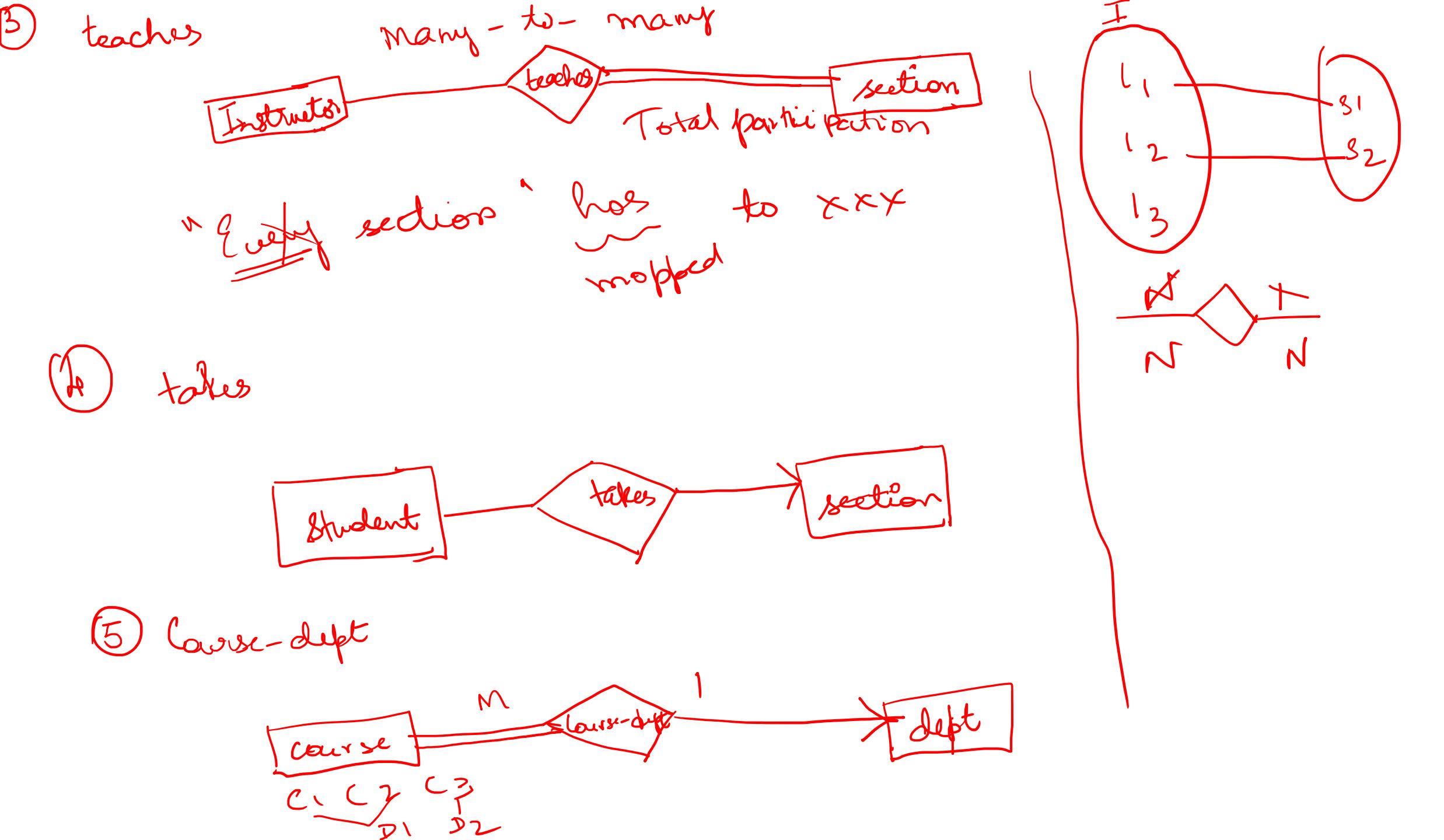
ER matches 100%  
with description.

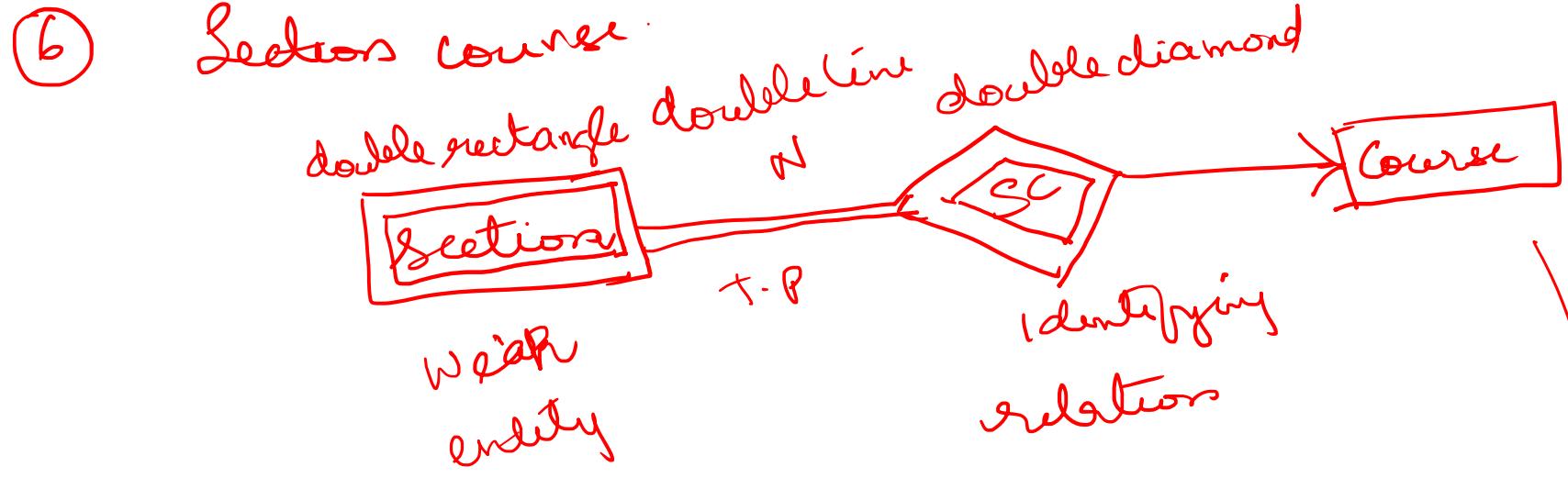
② Student - Department



Alternative

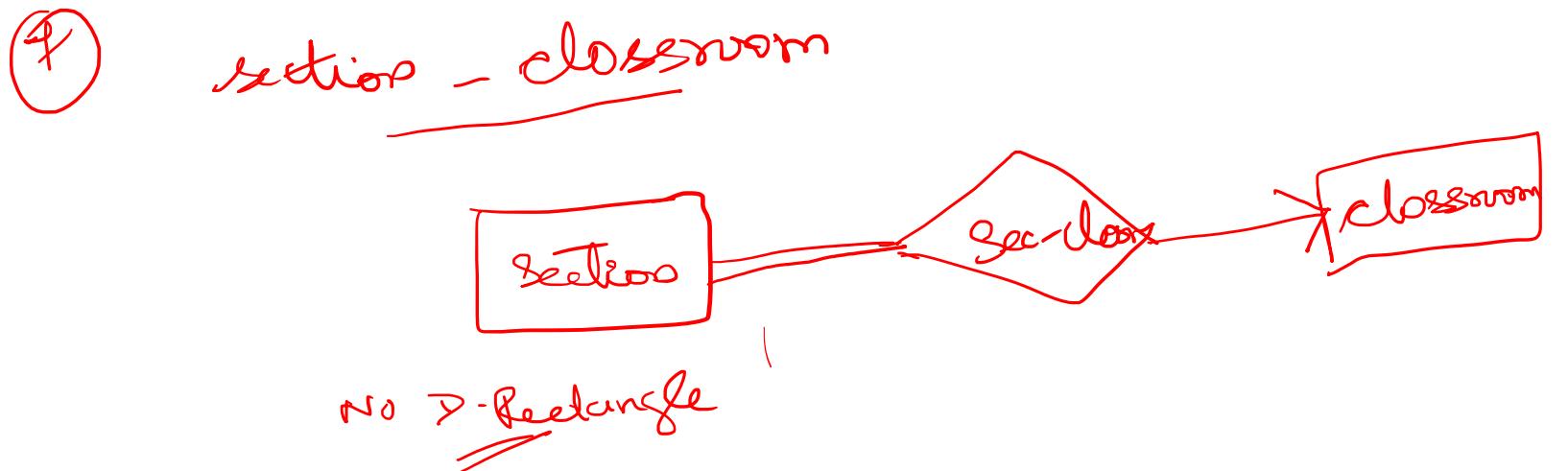






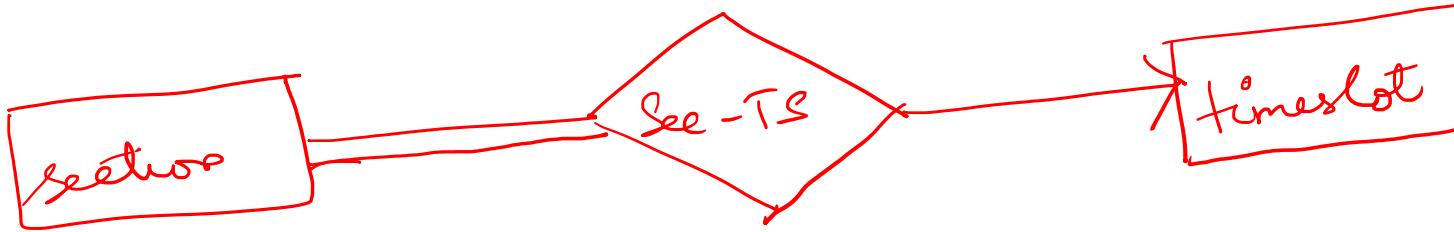
Courses are offered  
as sections

DB: → S<sub>1</sub> S<sub>2</sub>



⑧

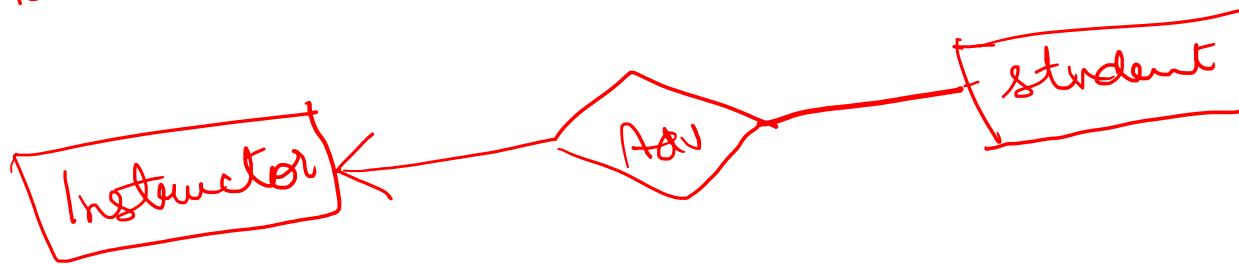
Section-timeslot



Wed 2:50 pm  
S1 → C1  
S2 → C2

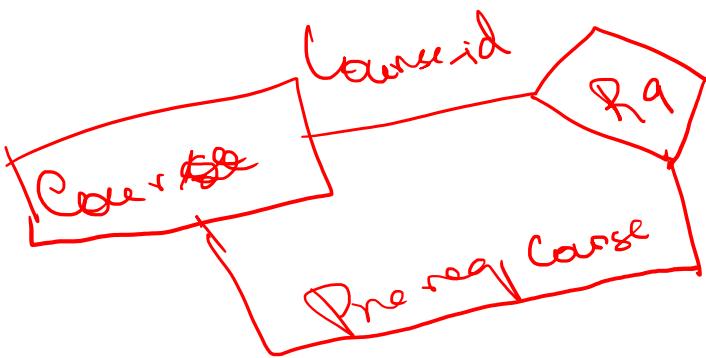
⑨

Adviser

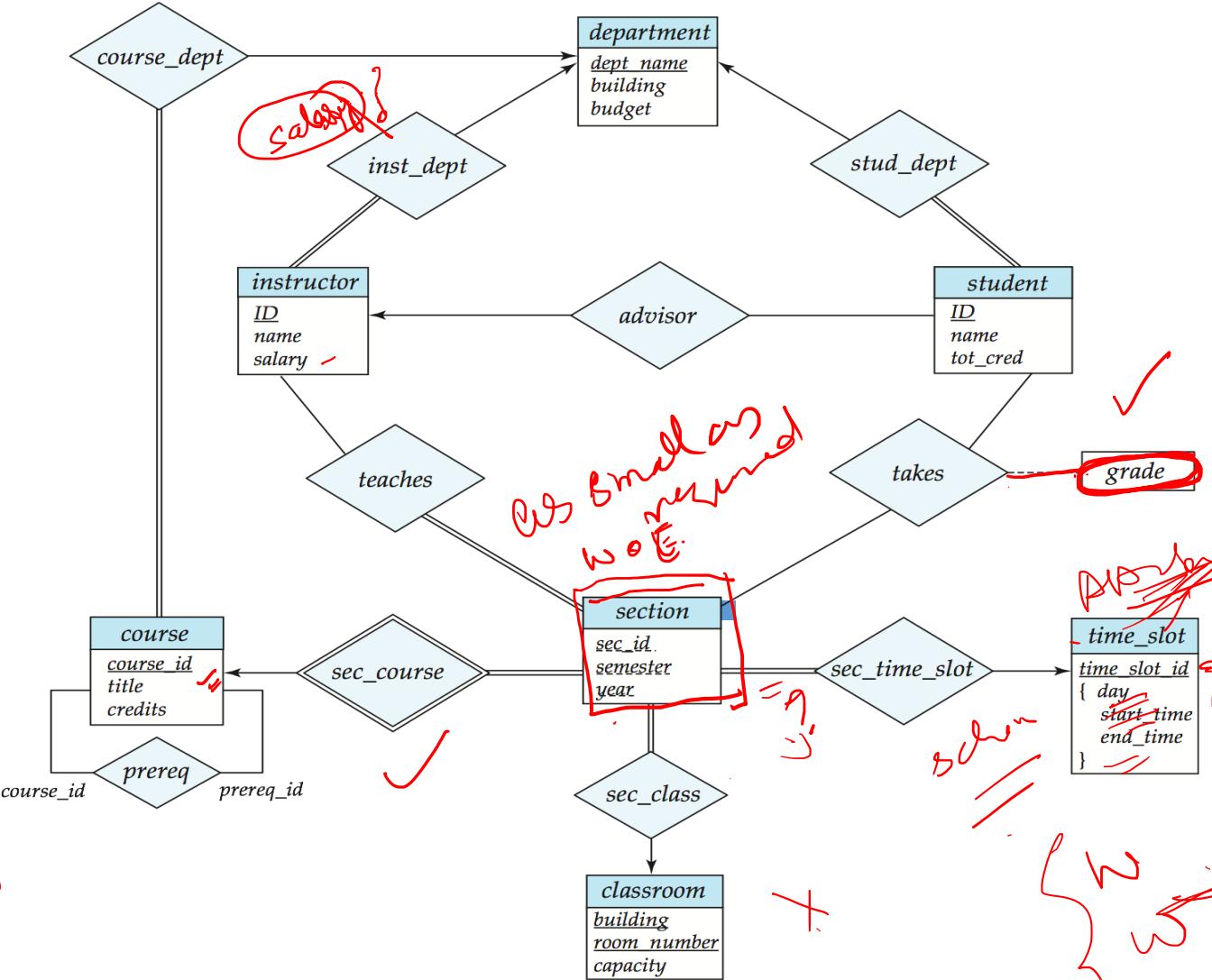


⑩

Course



# E-R Diagram for a University Enterprise



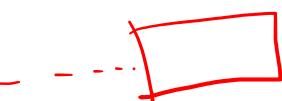
Note: ans in  
single form

Final work

Extra attribute  
etc. (some attr.)

1 - 1  
M - M

Attributes



{ Entity  
Relationship }

c1  
c2  
c3  
c4

W 2.00  
W 2.00  
W 2.00

- **E-R diagram** can express the overall logical structure of a database graphically.
- Basic structure
- cardinality mapping
- participation
- Weak entity
- Composite
- multivalued

[Entity-Relationship model: Chen notation example - YouTube](#)

# Construct ER diagram (student work)

1. Each person has at most one e-mail address. Each e-mail address belongs to exactly one person.
2. An apartment is located in a house in a street in a city in a country.
3. Two teams play football against each other. A referee makes sure the rules are followed.
4. Construct an E-R diagram for a hospital with a set of patients and a set of medical doctors. Associate with each patient a log of the various tests and examinations conducted.
5. Consider a company database having each employee works for only 1 department which is managed by one of them. Department has projects running under it and assigned to a team of employees. Further employees are managed by a supervisor. Company is also interested to know the details of employee's dependents.

D P - E

wo<sup>s</sup>

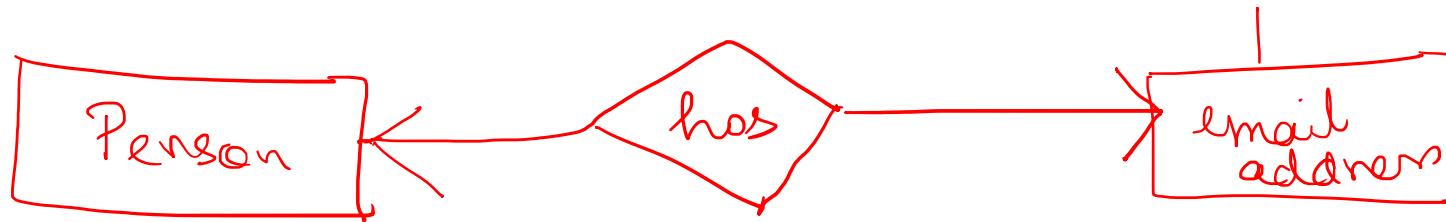
dept #

Each person has at most one e-mail address. Each e-mail address belongs to exactly one person.

== at least  
upto ..

min - - max  
= =

ER diagram



Attributes vs Entities ???

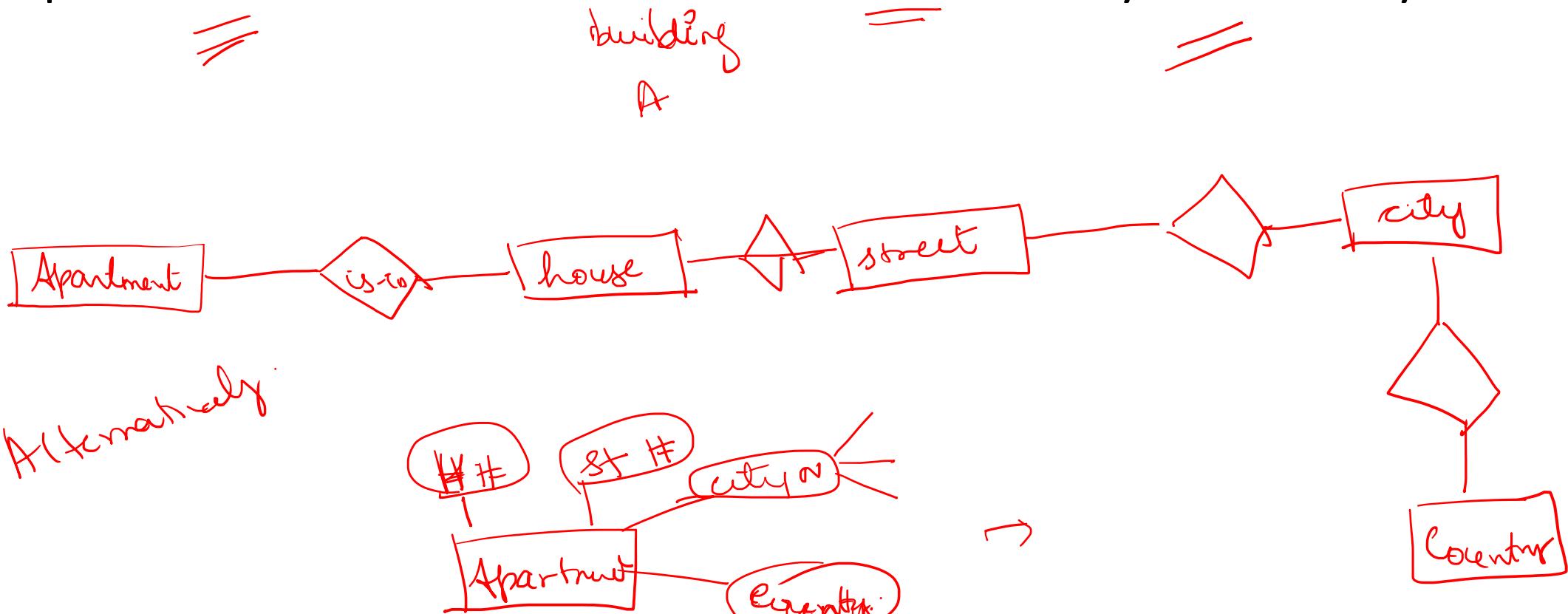
Object can express about its properties

Entity;

Ambiguous

Attributes Vs Entity

- An apartment is located in a house in a street in a city in a country.

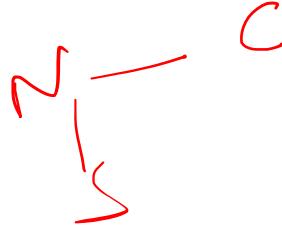


- Two teams play football against each other. A referee makes sure the rules are followed.

✓



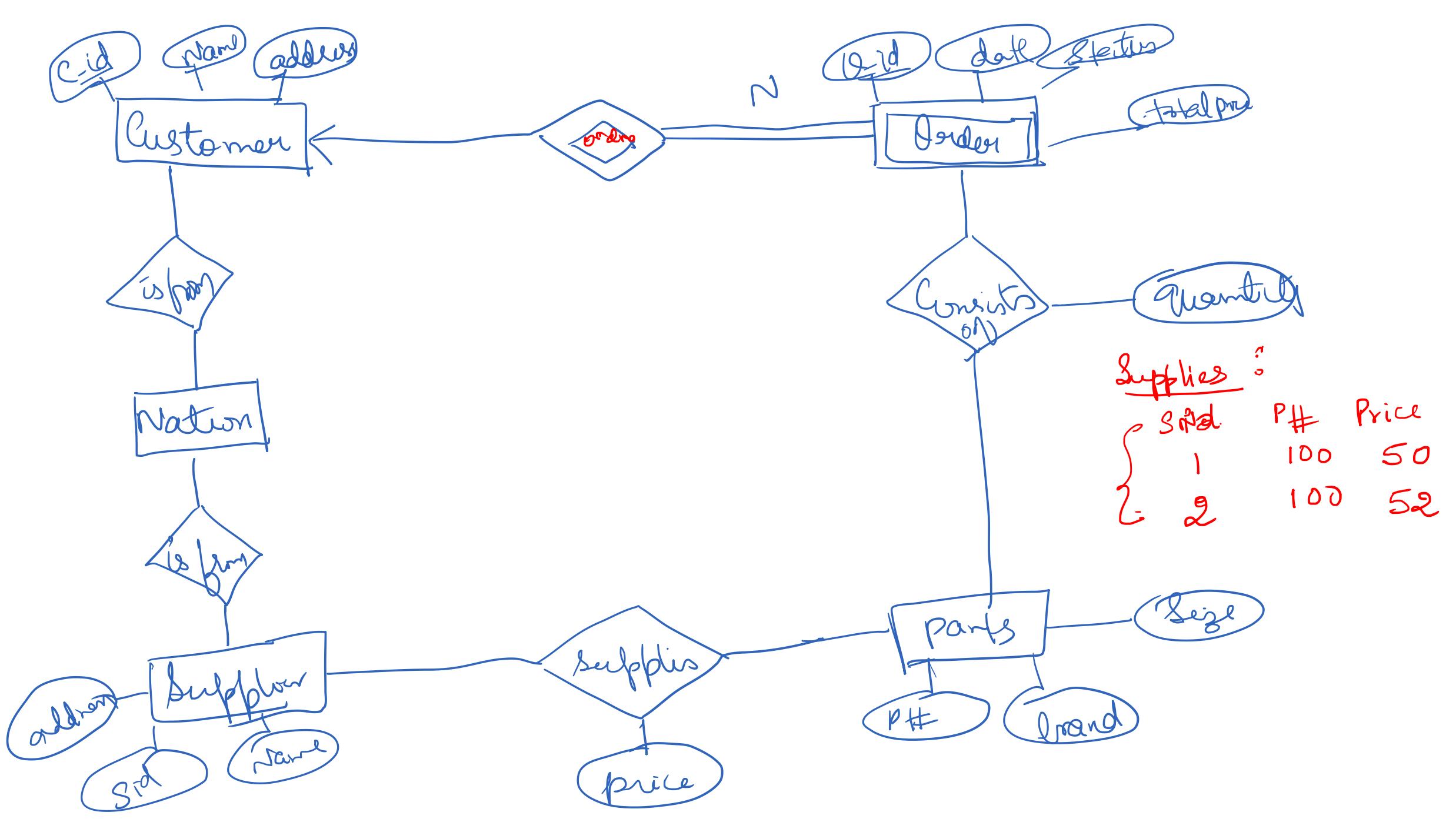
# examples

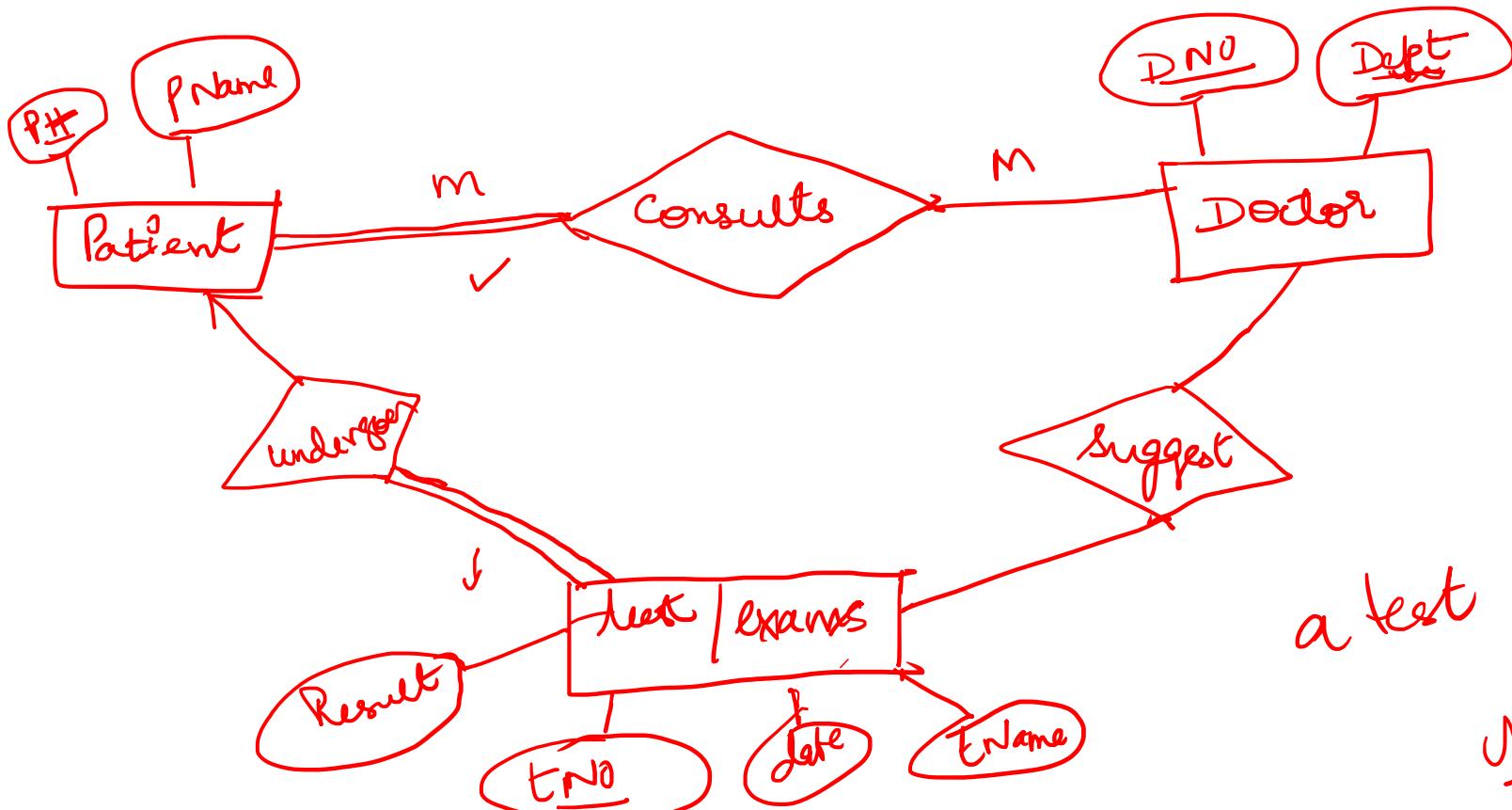


The wholesale supplier has customers that place orders, which are placed on a particular date and have a total price, current status, and an order number (starting from 1 for each customer).

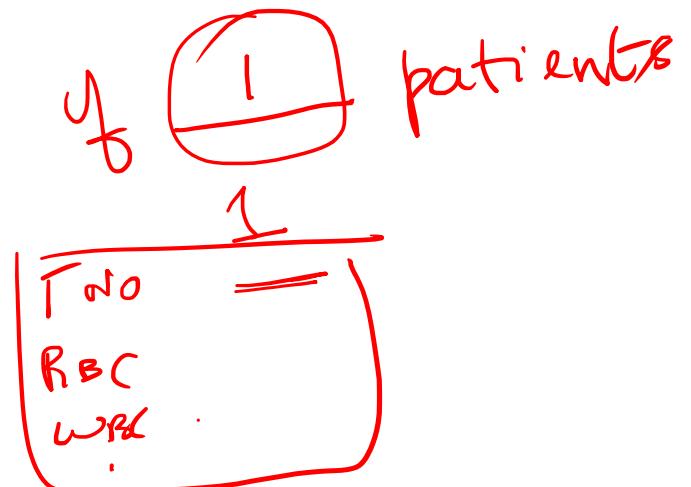
In each order, a customer can order several parts (products), each in a different quantity and at a (possibly discounted) price. We also want to model the date on which each of the parts has been sent. The parts are provided by suppliers. Each part may be provided by several suppliers and customers may order the same part of different suppliers in the same order, but in this case, they may have different (retail) prices. Customers and suppliers have a name, an address, a phone number, and a customer/supplier number and they come from a certain nation, which in turn is from a particular region (of the world).

Parts have a brand, a size, and a retail price.

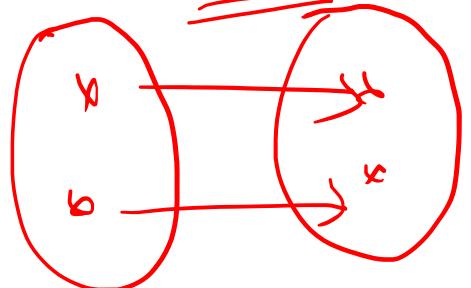


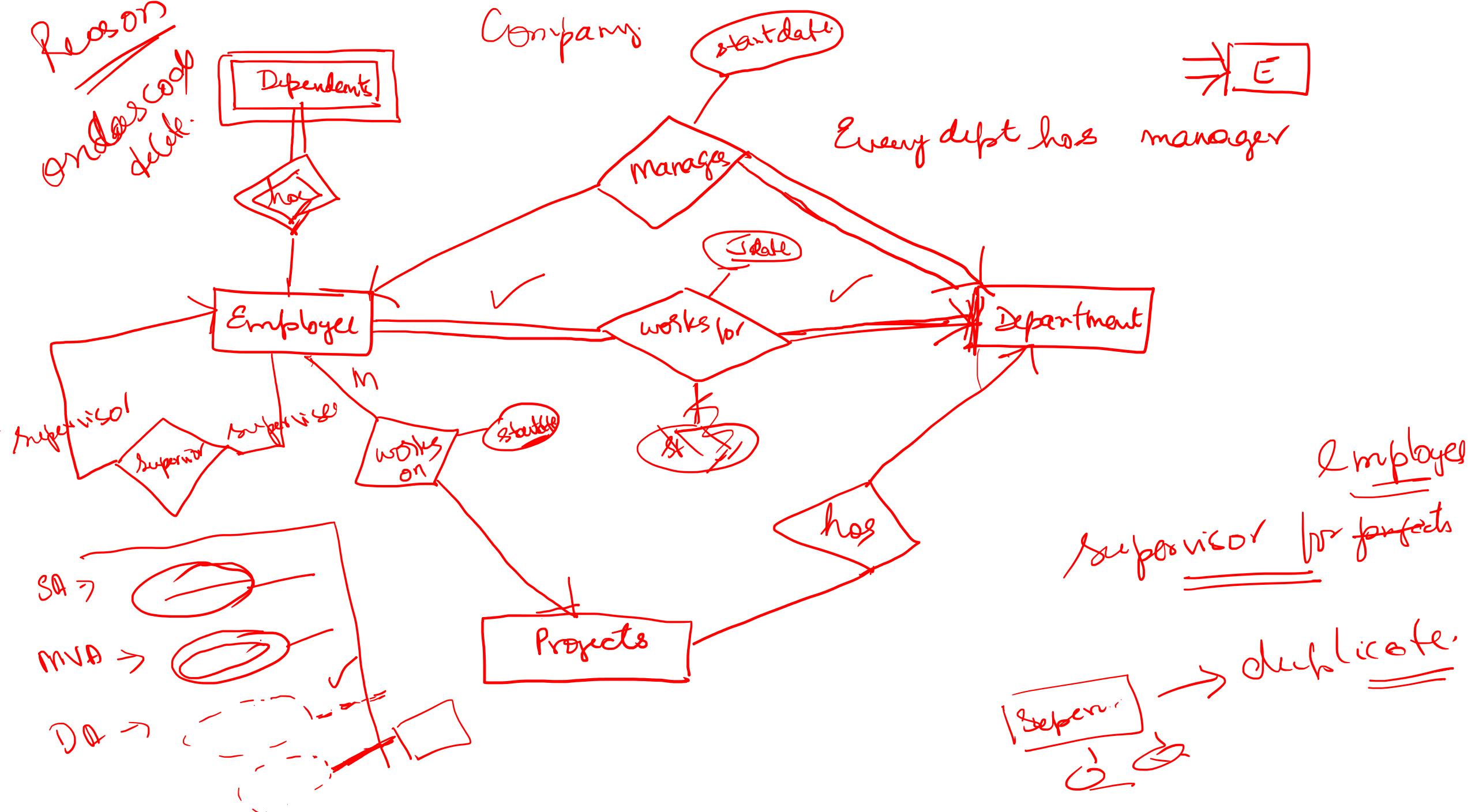


a test contains details



Set theory  
Relations







# Reduction to Relational Schemas

# Reduction to Relation Schemas

- Entity sets and relationship sets can be expressed uniformly as *relation schemas* that represent the contents of the database.
- A database which conforms to an E-R diagram can be represented by a collection of schemas.
- For each entity set and relationship set there is a unique schema that is assigned the name of the corresponding entity set or relationship set.
- Each schema has a number of columns (generally corresponding to attributes), which have unique names.

(Table Name)

**COURSE**

| Course_name               | Course_number | Credit_hours | Department |
|---------------------------|---------------|--------------|------------|
| Intro to Computer Science | CS1310        | 4            | CS         |
| Data Structures           | CS3320        | 4            | CS         |
| Discrete Mathematics      | MATH2410      | 3            | MATH       |
| Database                  | CS3380        | 3            | CS         |

**SECTION**

| Section_identifier | Course_number | Semester | Year | Instructor |
|--------------------|---------------|----------|------|------------|
| 85                 | MATH2410      | Fall     | 04   | King       |
| 92                 | CS1310        | Fall     | 04   | Anderson   |
| 102                | CS3320        | Spring   | 05   | Knuth      |
| 112                | MATH2410      | Fall     | 05   | Chang      |
| 119                | CS1310        | Fall     | 05   | Anderson   |
| 135                | CS3380        | Fall     | 05   | Stone      |

**GRADE\_REPORT**

| Student_number | Section_identifier | Grade |
|----------------|--------------------|-------|
| 17             | 112                | B     |
| 17             | 119                | C     |
| 8              | 85                 | A     |
| 8              | 92                 | A     |
| 8              | 102                | B     |
| 8              | 135                | A     |

**PREREQUISITE**

| Course_number | Prerequisite_number |
|---------------|---------------------|
| CS3380        | CS3320              |
| CS3380        | MATH2410            |
| CS3320        | CS1310              |

*Table*

*attribute*

*tuples*

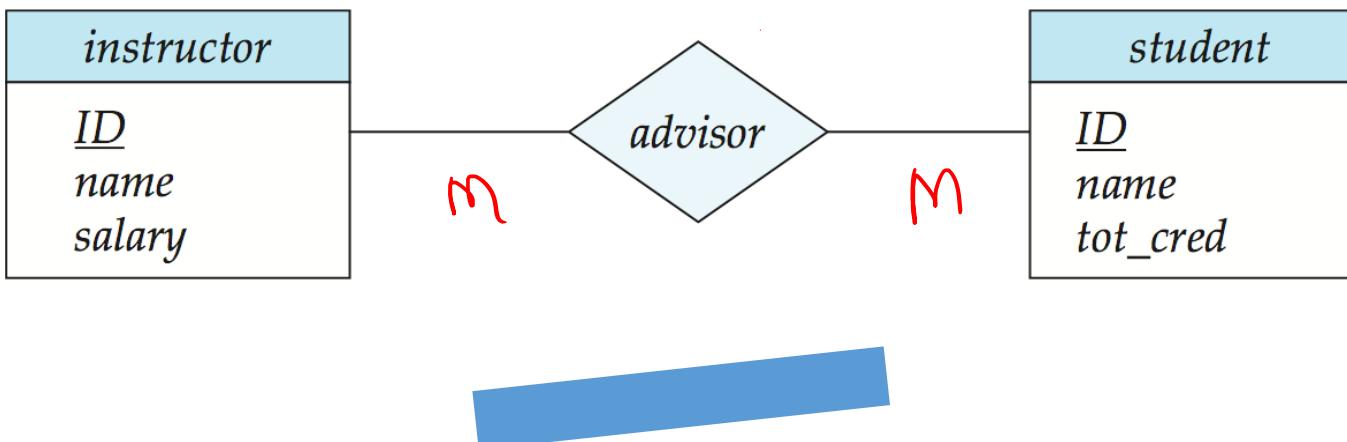
**Figure 1.2**  
A database that stores student and course information.

# Representing Relationship Sets

Rule 1:

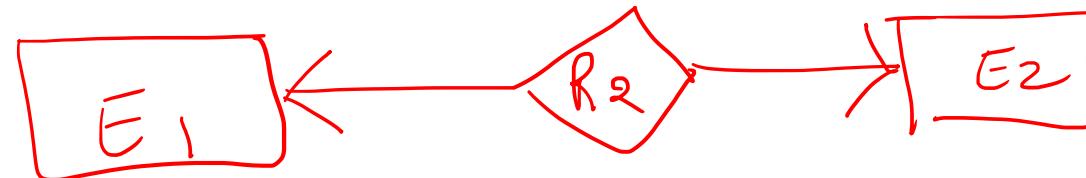
- A many-to-many relationship set is represented as a schema with attributes for the primary keys of the two participating entity sets, and any descriptive attributes of the relationship set.
- Example: schema for relationship set *advisor*

advisor ~~=~~ (s\_id, i\_id)





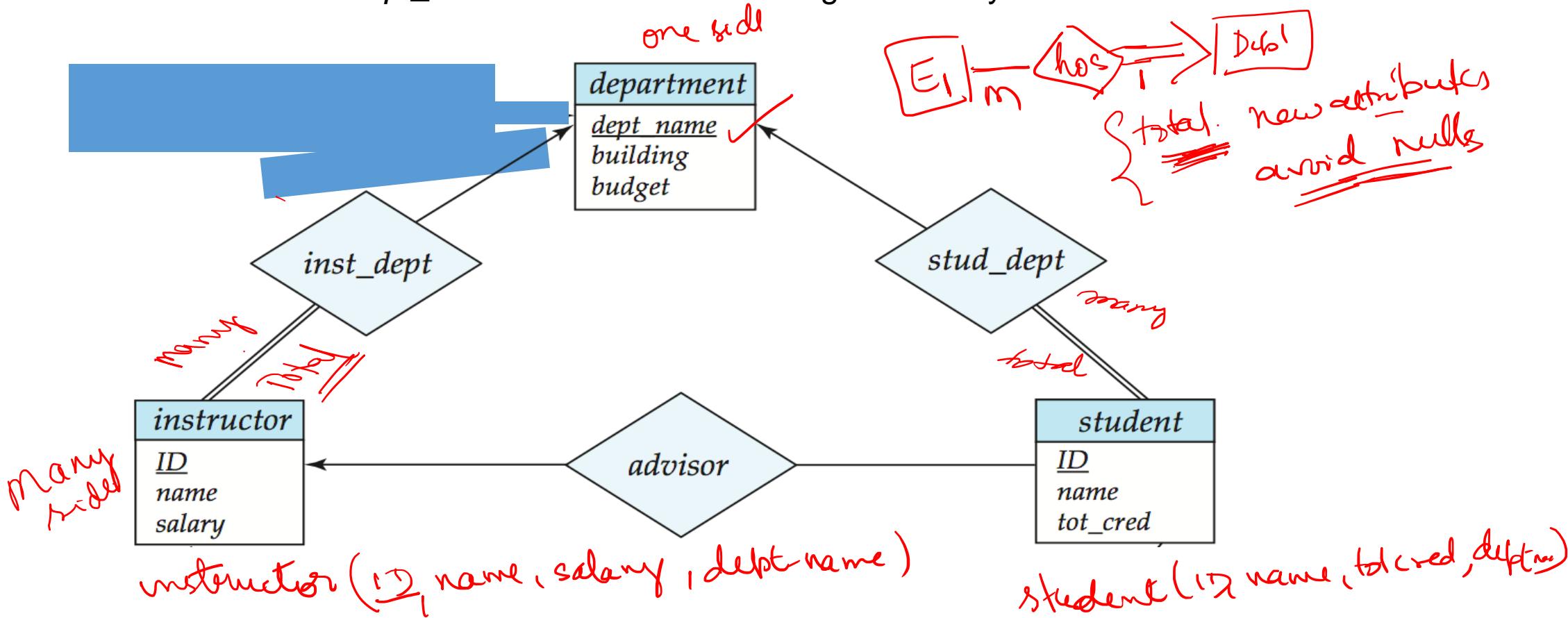
- For **one-to-one** relationship sets, either side can be chosen to act as the “many” side
- That is, **extra attribute** can be added to either of the tables corresponding to the two entity sets



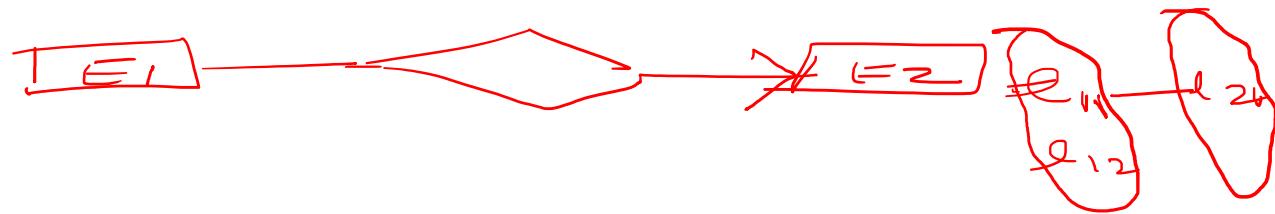
1 - 1 as  
well

### Rule 2

- Many-to-one and one-to-many relationship sets that are total on the many-side can be represented by adding an extra attribute to the "many" side, containing the primary key of the "one" side
- If Partial participation: Separate table *for the partial relation*
- Example: Instead of creating a schema for relationship set *inst\_dept*, add an attribute *dept\_name* to the schema arising from entity set *instructor*



# ~~Redundancy~~<sup>Redundancy</sup> of Schemas (Cont.)



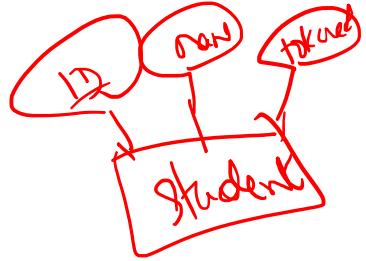
$E_1 \rightarrow E_2$  ID, name,  $E_2 \rightarrow$  DID  
do. 0000 20  
002 100 null 3

# issues

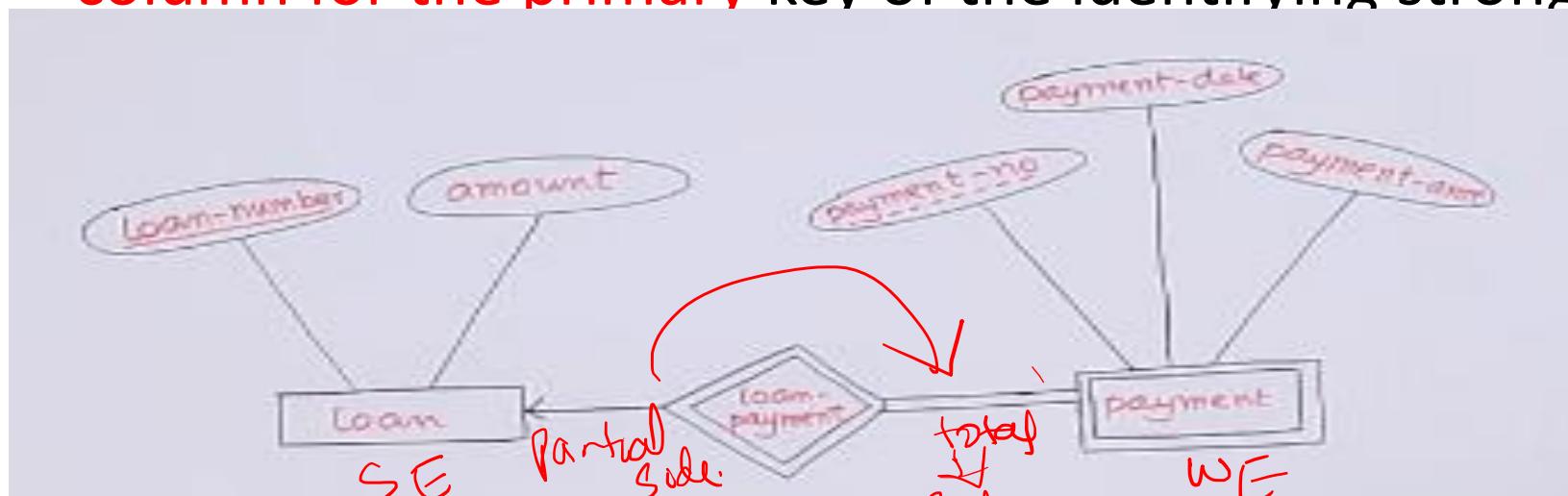
- If participation is *partial* on the “many” side, replacing a schema by an extra attribute in the schema corresponding to the “many” side could result in null values
- The **schema corresponding** to a **relationship set linking a weak entity set to its identifying strong entity set** is **redundant**.
  - Example: The *section* schema already contains the attributes that would appear in the *sec\_course* schema

# Representing Entity Sets With Simple Attributes

Rule 4:



- A strong entity set reduces to a schema with the same attributes  
 $student(\underline{ID}, \underline{name}, \underline{tot\_cred})$
- A weak entity set becomes a table that includes a column for the primary key of the identifying strong



$\underline{\text{loan}}(\underline{\text{loan\#}}, \underline{\text{amt}})$

Rule 2 - WE  
 $\underline{\text{payment}}(\underline{\text{loan\#}}, \underline{\text{pay\#}}, \underline{\text{date}}, \underline{x})$

# Composite and Multivalued Attributes

Rules

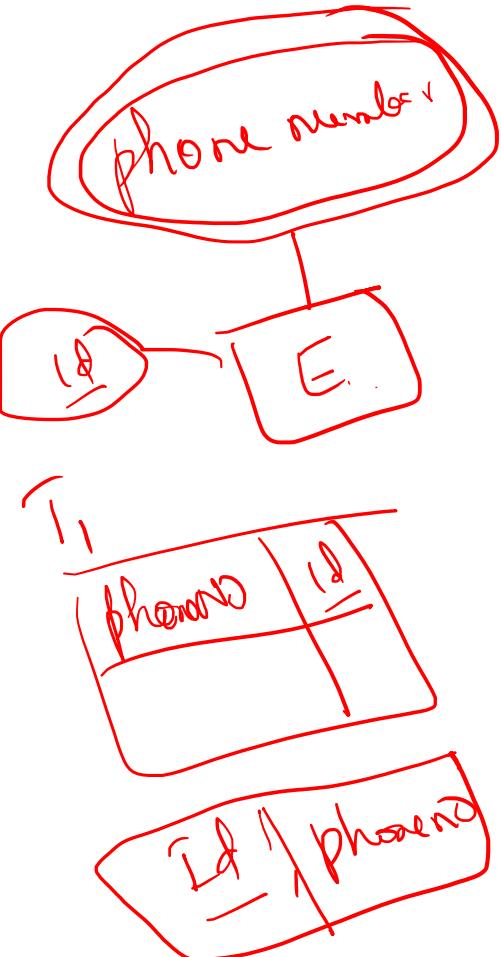
| instructor         |  |
|--------------------|--|
| <u>ID</u>          |  |
| <u>name</u>        |  |
| first_name         |  |
| middle_initial     |  |
| last_name          |  |
| <u>address</u>     |  |
| street             |  |
| street_number      |  |
| street_name        |  |
| apt_number  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

- Composite attributes are flattened out by creating a separate attribute for each component attribute
  - Example: given entity set *instructor* with composite attribute *name* with component attributes *first\_name* and *last\_name* the schema corresponding to the entity set has two attributes *name\_first\_name* and *name\_last\_name*
    - Prefix omitted if there is no ambiguity
- Ignoring multivalued attributes, extended instructor schema is
  - *instructor(ID, first\_name, middle\_initial, last\_name, street\_number, street\_name, apt\_number, city, state, zip\_code, date\_of\_birth)*

# Composite and Multivalued Attributes

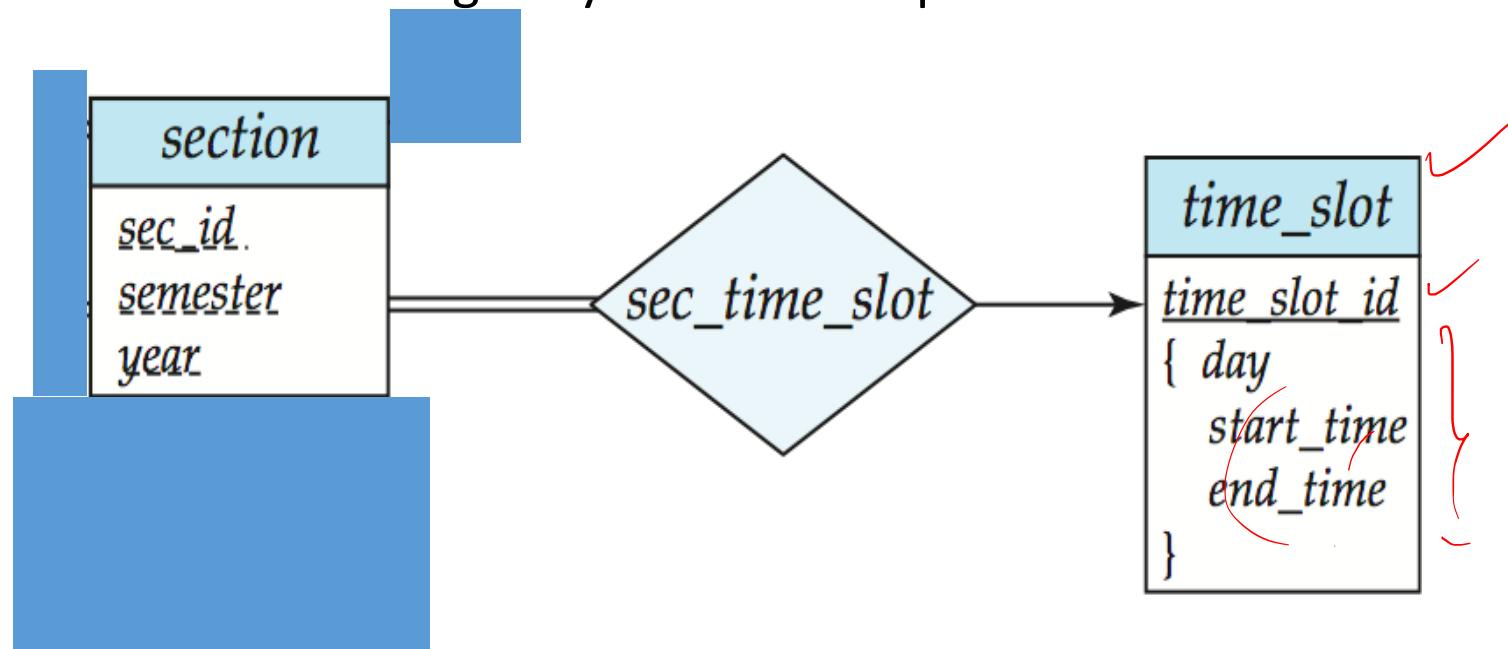
Rule 6

- A multivalued attribute  $M$  of an entity  $E$  is \_\_\_\_\_ represented by a separate schema  $EM$ 
  - Schema  $EM$  has attributes corresponding to the primary key of  $E$  and an attribute corresponding to multivalued attribute  $M$
  - Example: Multivalued attribute  $phone\_number$  of  $instructor$  is represented by a schema:  
 $inst\_phone = ( \underline{ID}, \underline{phone\_number} )$
  - Each value of the multivalued attribute maps to a separate tuple of the relation on schema  $EM$ 
    - For example, an  $instructor$  entity with primary key 22222 and phone numbers 456-7890 and 123-4567 maps to two tuples:  
(22222, 456-7890) and (22222, 123-4567)



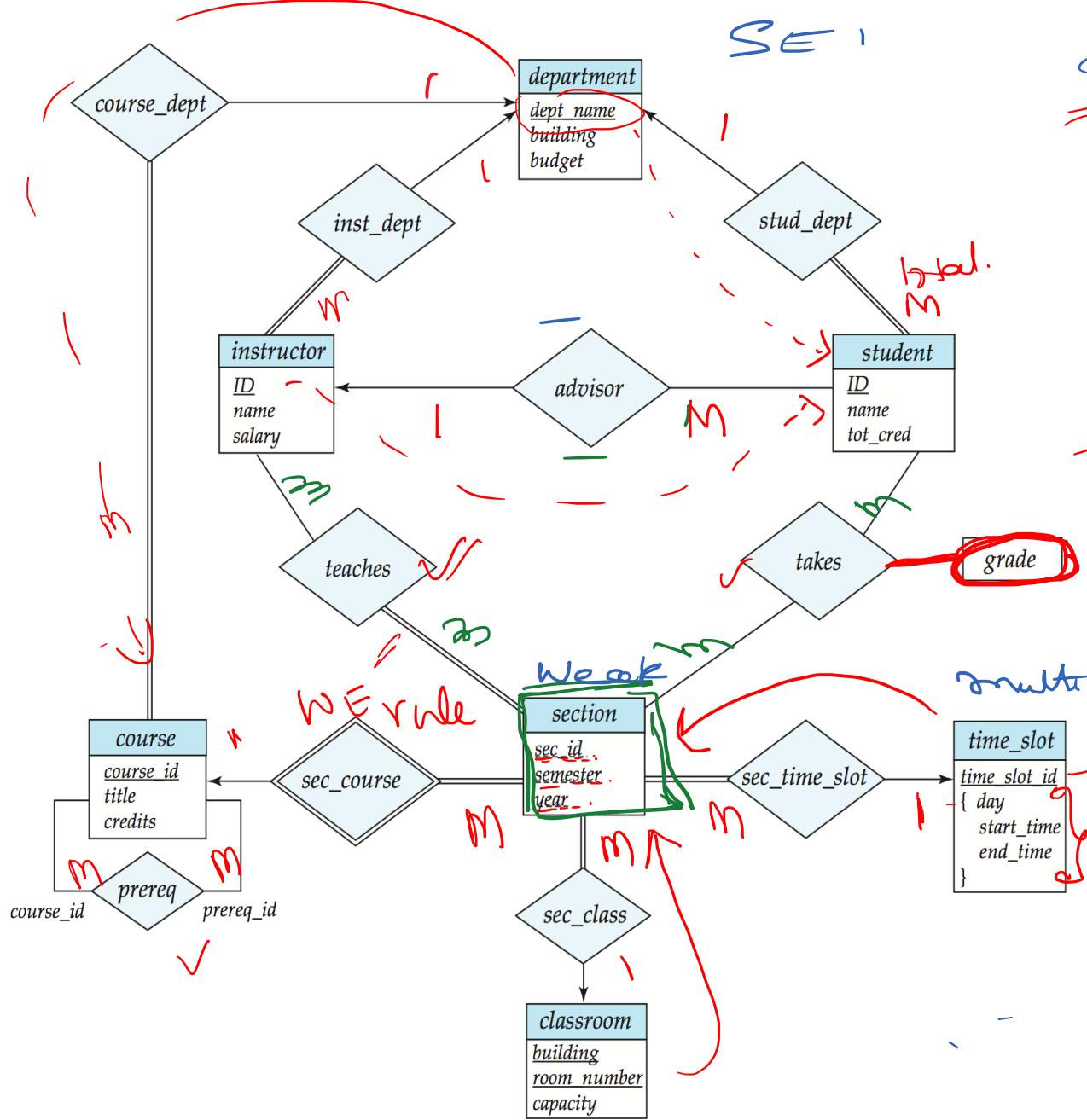
# Multivalued Attributes (Cont.)

- Special case: entity ***time\_slot*** has only one attribute other than the primary-key attribute, and that **attribute is multivalued**
  - Optimization: Don't create the relation corresponding to the entity, just create the one corresponding to the multivalued attribute
  - *time\_slot(time\_slot\_id, day, start\_time, end\_time)*
  - Caveat: *time\_slot* attribute of *section* (from *sec\_time\_slot*) cannot be a foreign key due to this optimization



# Summarizing the rule

- Regular entity: schema with attributes
- Weak entities: schema including strong entity PK
- Composite attribute: schema is extended
- Multivalued attribute: new schema with its individual attribute and PK of the parent table
- Based on degree
  - One to one: separate schema + PK of either of the entity on other side
  - Many to one or one to many: schema for many side include PK of one side
  - Many to many: relationship is converted to schema
- Based on participation:
  - Total participation: on many side add PK of one side
  - Partial participation: separate table



SE'

SE: dept  
instructor  
class room  
course  
student

NE: section

inv: timeslot

many-many  $\rightarrow$  PK of E2

multi:

teachers

takes

{ MU }

attends

schema



dept(dName, building, budget)      ↗ m:1 mapping

student(ID, name, bDate, dName, I-ID)

Instructor(I-ID, name, salary, dName)      ↗ m:1 mapping  
originally from Instructor

classroom(building, room#, capacity)

course(CourseID, title, credit, dName)      ↗ m:1 mapping

Due to m:n mapping

teaches (I-ID,  
PK of instructor, selid, courseid, sem, year)  
PK of section

takes ( ID, selid, courseid, sem, year, grade)  
PK of student PK of section attribute of relation

prereq ( courseid, prereqid)

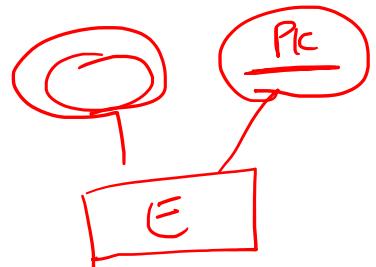
Weak entity Sections

Sections

secid, sem, year, courseid, building, room#, timeSlotid )  
W.E.guide  
1 : M rule

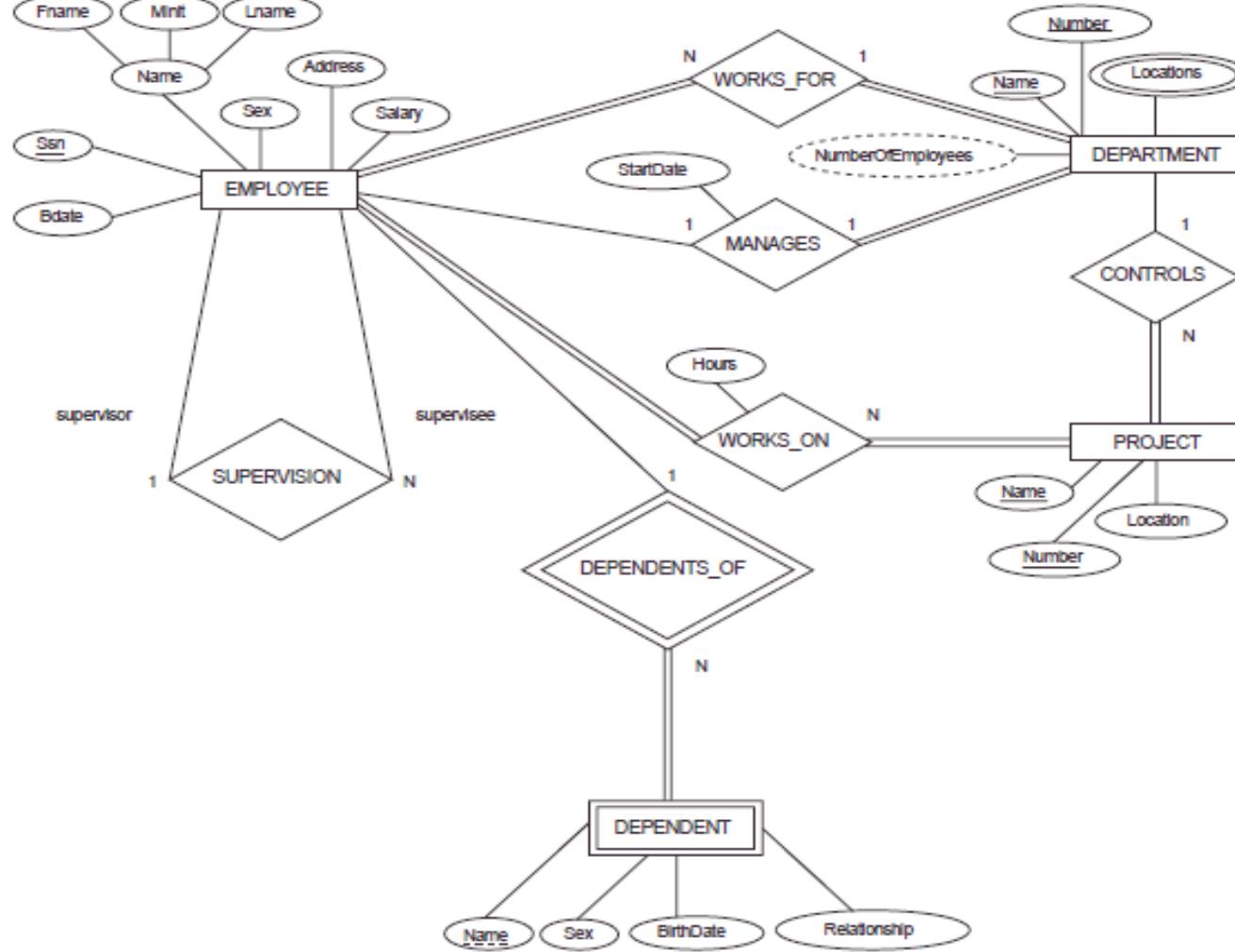
M-V. attribute

timeslot details( start, end, day, timeslotid)



Advisor (ID, ID)

*classroom*

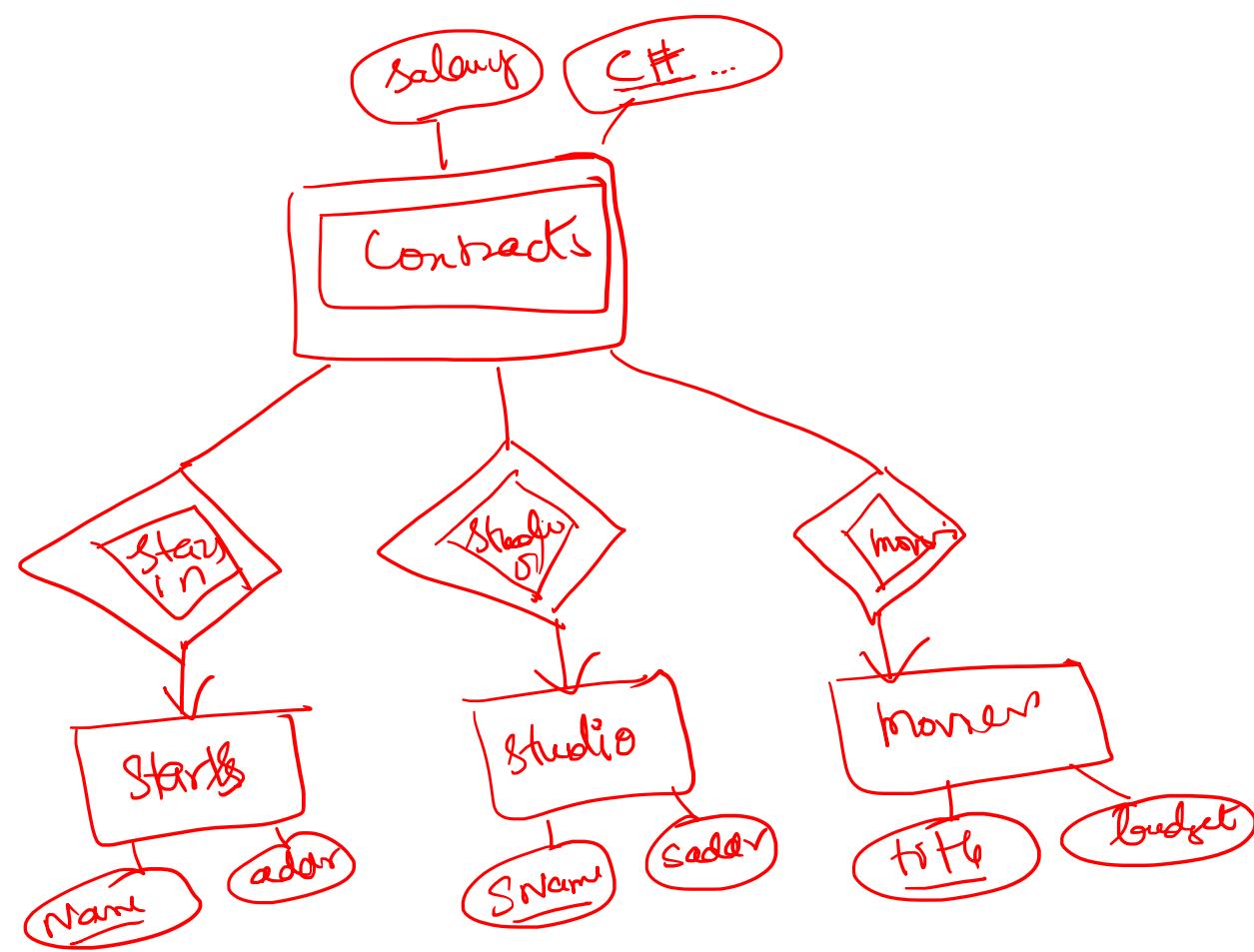


stars(Name, address)

studio(SName, address)

Movie(Title, Budget)

Contracts(C#, Salary,  
Name, SName, Title)



sunitha @ manipal . edu

# Design-phase 2

part1



## Database Management Systems

# Module 16: Relational Database Design/1



**Partha Pratim Das**

*Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur*

[ppd@cse.iitkgp.ernet.in](mailto:ppd@cse.iitkgp.ernet.in)

Srijoni Majumdar  
Himadri B G S Bhuyan  
Gurunath Reddy M



---

**Database System Concepts, 6<sup>th</sup> Ed.**

©Silberschatz, Korth and Sudarshan  
[www.db-book.com](http://www.db-book.com)

# Week 03

## Recap

PPD

### ■ **Module 11: Advanced SQL**

- Accessing SQL From a Programming Language
- Functions and Procedural Constructs
- Triggers

### ■ **Module 12: Formal Relational Query Languages**

- Relational Algebra
- Tuple Relational Calculus (Overview only)
- Domain Relational Calculus (Overview only)
- Equivalence of Algebra and Calculus

### ■ **Module 13: Entity-Relationship Model/1**

- Design Process
- E-R Model

### ■ **Module 14: Entity-Relationship Model/2**

- E-R Diagram
- E-R Model to Relational Schema

### ■ **Module 15: Entity-Relationship Model/3**

- Extended E-R Features
- Design Issues

# Module Objectives

- To identify the features of good relational design
- To familiarize with the First Normal Form
- To Introduce Functional Dependencies

NPTEL

# Module Outline

PPD

- Features of Good Relational Design
- Atomic Domains and First Normal Form
- Functional Dependencies

- **Features of Good Relational Design**
- Atomic Domains and First Normal Form
- Functional Dependencies

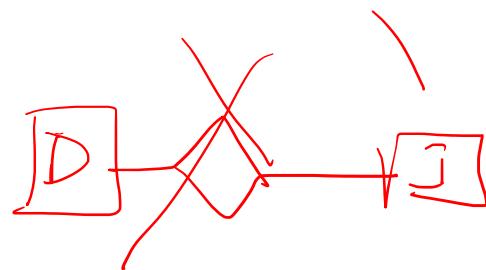
## FEATURES OF GOOD RELATIONAL DESIGN

# Combine Schemas?

- Suppose we combine *instructor* and *department* into *inst\_dept*
  - (No connection to relationship set *inst\_dept*)
- Result is possible repetition of information (*building* and *budget* against *dept\_name*)

| ID    | name       | salary | dept_name  | building | budget |
|-------|------------|--------|------------|----------|--------|
| 22222 | Einstein   | 95000  | Physics    | Watson   | 70000  |
| 12121 | Wu         | 90000  | Finance    | Painter  | 120000 |
| 32343 | El Said    | 60000  | History    | Painter  | 50000  |
| 45565 | Katz       | 75000  | Comp. Sci. | Taylor   | 100000 |
| 98345 | Kim        | 80000  | Elec. Eng. | Taylor   | 85000  |
| 76766 | Crick      | 72000  | Biology    | Watson   | 90000  |
| 10101 | Srinivasan | 65000  | Comp. Sci. | Taylor   | 100000 |
| 58583 | Califieri  | 62000  | History    | Painter  | 50000  |
| 83821 | Brandt     | 92000  | Comp. Sci. | Taylor   | 100000 |
| 15151 | Mozart     | 40000  | Music      | Packard  | 80000  |
| 33456 | Gold       | 87000  | Physics    | Watson   | 70000  |
| 76543 | Singh      | 80000  | Finance    | Painter  | 120000 |

An answer  
insert  
delete  
update →



update.

New info.

Physics  
→

Taylor : 90000

update

# A Combined Schema Without Repetition

- Consider combining relations
  - $\text{sec\_class(sec\_id, building, room\_number)}$  and
  - $\text{section(course\_id, sec\_id, semester, year)}$
- into one relation
  - $\text{section(course\_id, sec\_id, semester, year, building, room\_number)}$  ↗
- No repetition in this case

# What About Smaller Schemas?

- Suppose we had started with *inst\_dept*. How would we know to split up (**decompose**) it into *instructor* and *department*?
- Write a rule “if there were a schema (*dept\_name, building, budget*), then *dept\_name* would be a candidate key”
- Denote as a **functional dependency**:  
$$\text{dept\_name} \rightarrow \text{building, budget}$$
- In *inst\_dept*, because *dept\_name* is not a candidate key, the building and budget of a department may have to be repeated.
  - This indicates the need to decompose *inst\_dept*
- Not all decompositions are good. Suppose we decompose *employee*(*ID, name, street, city, salary*) into
  - employee1* (*ID, name*)
  - employee2* (*name, street, city, salary*)
- The next slide shows how we lose information -- we cannot reconstruct the original *employee* relation -- and so, this is a **lossy decomposition**.



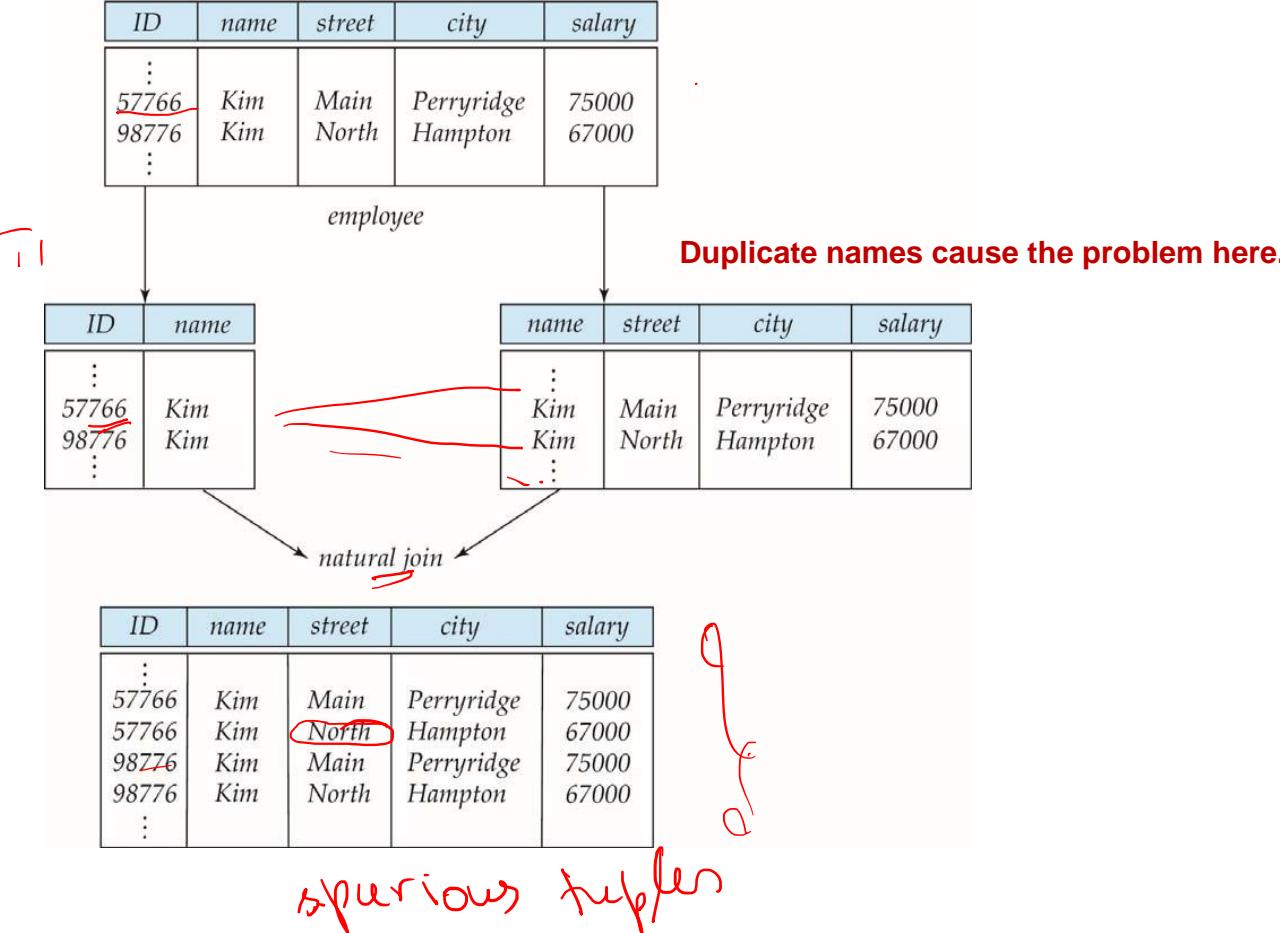
1  
6  
.1  
0  
a  
t

@  
Sb  
ia  
Is  
be  
eS  
ry  
ss  
ct  
he  
am  
tC  
zo  
rn  
tc  
he  
ap  
nt  
dS  
S-  
u6  
dE  
ad  
ri  
so  
hn  
an

SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

# A Lossy Decomposition

PPD



1  
6

.

1

b

a

@

Sa

i b

l a

bS

ee

rS

sy

cS

ht

aE

tM

zC

'o

Kn

oc

re

tp

ht

as

n\_

d6

S<sub>h</sub>

u<sup>E</sup>

u<sub>d</sub>

di

ai

r<sup>O</sup>

s

h

a

## Example of Lossless-Join Decomposition

- **Lossless join decomposition**
- Decomposition of  $R = (A, B, C)$

$$R_1 = (A, B) \quad R_2 = (B, C)$$

*Main*

| A        | B | C |
|----------|---|---|
| $\alpha$ | 1 | A |
| $\beta$  | 2 | B |

$r$

*$R_1$*

| A        | B |
|----------|---|
| $\alpha$ | 1 |
| $\beta$  | 2 |

$\Pi_{A,B}(r)$

*$R_2$*

| B | C |
|---|---|
| 1 | A |
| 2 | B |

$\Pi_{B,C}(r)$

$$\Pi_A(r) \bowtie \Pi_B(r)$$

| A        | B | C |
|----------|---|---|
| $\alpha$ | 1 | A |
| $\beta$  | 2 | B |

*Main*

No loss + no spurious tuples  
are seen

## PPD

- Features of Good Relational Design
- **Atomic Domains and First Normal Form**
- Functional Dependencies

# ATOMIC DOMAINS AND FIRST NORMAL FORM

# First Normal Form (1NF)

- Domain is **atomic** if its elements are considered to be indivisible units
  - Examples of non-atomic domains:
    - Set of names, composite attributes
    - Identification numbers like CS101 that can be broken up into parts
      - A relational schema R is in **first normal form** if
        - the domains of all attributes of R are atomic
    - the value of each attribute contains only a single value from that domain
  - Non-atomic values complicate storage and encourage redundant (repeated) storage of data
    - Example: Set of accounts stored with each customer, and set of owners stored with each account
    - We assume all relations are in first normal form

# First Normal Form (Cont'd)

- Atomicity is actually a property of how the elements of the domain are used
  - Example: Strings would normally be considered indivisible
  - Suppose that students are given roll numbers which are strings of the form *CS0012* or *EE1127*
  - If the first two characters are extracted to find the department, the domain of roll numbers is not atomic
  - Doing so is a bad idea: leads to encoding of information in application program rather than in the database

# First Normal Form (Cont'd)

- The following is not in 1NF

**Customer**

| Customer ID | First Name | Surname | Telephone Number                     |
|-------------|------------|---------|--------------------------------------|
| 123         | Pooja      | Singh   | 555-861-2025, 192-122-1111           |
| 456         | San        | Zhang   | (555) 403-1659 Ext. 53; 182-929-2929 |
| 789         | John       | Doe     | 555-808-9633                         |

- A telephone number is composite
- Telephone number is multi-valued

Source: [https://en.wikipedia.org/wiki/First\\_normal\\_form](https://en.wikipedia.org/wiki/First_normal_form)

# First Normal Form (Cont'd)

- Consider:

**Customer**

| Customer ID | First Name | Surname | Telephone Number1      | Telephone Number2 |
|-------------|------------|---------|------------------------|-------------------|
| 123         | Pooja      | Singh   | 555-861-2025           | 192-122-1111      |
| 456         | San        | Zhang   | (555) 403-1659 Ext. 53 | 182-929-2929      |
| 789         | John       | Doe     | 555-808-9633           |                   |

- Is in 1NF if telephone number is not considered composite
- However, conceptually, we have two attributes for the same concept
  - Arbitrary and meaningless ordering of attributes
  - How to search telephone numbers
  - Why only two numbers?

Source: [https://en.wikipedia.org/wiki/First\\_normal\\_form](https://en.wikipedia.org/wiki/First_normal_form)

# First Normal Form (Cont'd)

- Is the following in 1NF?

| Customer    |            |         |                        |
|-------------|------------|---------|------------------------|
| Customer ID | First Name | Surname | Telephone Number       |
| 123         | Pooja      | Singh   | 555-861-2025           |
| 123         | Pooja      | Singh   | 192-122-1111           |
| 456         | San        | Zhang   | 182-929-2929           |
| 456         | San        | Zhang   | (555) 403-1659 Ext. 53 |
| 789         | John       | Doe     | 555-808-9633           |

- Duplicated information
- ID is no more the key. Key is (ID, Telephone Number)

Source: [https://en.wikipedia.org/wiki/First\\_normal\\_form](https://en.wikipedia.org/wiki/First_normal_form)

# First Normal Form (Cont'd)

- Better to have 2 relations:

| Customer Name      |            |         | Customer Telephone Number |                         |
|--------------------|------------|---------|---------------------------|-------------------------|
| <u>Customer ID</u> | First Name | Surname | <u>Customer ID</u>        | <u>Telephone Number</u> |
| 123                | Pooja      | Singh   | 123                       | 555-861-2025            |
| 456                | San        | Zhang   | 123                       | 192-122-1111            |
| 789                | John       | Doe     | 456                       | (555) 403-1659 Ext. 53  |
|                    |            |         | 456                       | 182-929-2929            |
|                    |            |         | 789                       | 555-808-9633            |

- One-to-Many relationship between parent and child relations

Source: [https://en.wikipedia.org/wiki/First\\_normal\\_form](https://en.wikipedia.org/wiki/First_normal_form)

## PPD

- Features of Good Relational Design
- Atomic Domains and First Normal Form
- **Functional Dependencies**

# FUNCTIONAL DEPENDENCIES

# Goal – Devise a Theory for the Following

- Decide whether a particular relation  $R$  is in “good” form.
- In the case that a relation  $R$  is not in “good” form, decompose it into a set of relations  $\{R_1, R_2, \dots, R_n\}$  such that
  - each relation is in good form
  - the decomposition is a lossless-join decomposition
- Our theory is based on:
  - functional dependencies
  - multivalued dependencies

# Functional Dependencies

- Constraints on the set of legal relations
- Require that the value for a certain set of attributes determines uniquely the value for another set of attributes
- A functional dependency is a generalization of the notion of a key

# Functional Dependencies (Cont.)

- Let  $R$  be a relation schema

$$\alpha \subseteq R \text{ and } \beta \subseteq R$$

- The **functional dependency**

$$\alpha \rightarrow \beta$$

**holds on**  $R$  if and only if for any legal relations  $r(R)$ , whenever any two tuples  $t_1$  and  $t_2$  of  $r$  agree on the attributes  $\alpha$ , they also agree on the attributes  $\beta$ . That is,

$$t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$$

- Example: Consider  $r(A,B)$  with the following instance of  $r$ .

|   |   |
|---|---|
| 1 | 4 |
| 1 | 5 |
| 3 | 7 |

- On this instance,  $A \rightarrow B$  does **NOT** hold, but  $B \rightarrow A$  does hold.

# Functional Dependencies (Cont.)

- $K$  is a superkey for relation schema  $R$  if and only if  $K \rightarrow R$
- $K$  is a candidate key for  $R$  if and only if
  - $K \rightarrow R$ , and
  - for no  $\alpha \subset K$ ,  $\alpha \rightarrow R$
- Functional dependencies allow us to express constraints that cannot be expressed using superkeys. Consider the schema:

*inst\_dept (ID, name, salary, dept\_name, building, budget)*. We expect these functional dependencies to hold:

$$\text{dept\_name} \rightarrow \text{building}$$

$$\text{and } ID \rightarrow \text{building}$$

but would not expect the following to hold:

$$\text{dept\_name} \rightarrow \text{salary}$$

# Use of Functional Dependencies

- We use functional dependencies to:
  - test relations to see if they are legal under a given set of functional dependencies.
    - If a relation  $r$  is legal under a set  $F$  of functional dependencies, we say that  $r$  **satisfies**  $F$
  - specify constraints on the set of legal relations
    - We say that  $F$  **holds on**  $R$  if all legal relations on  $R$  satisfy the set of functional dependencies  $F$
- Note: A specific instance of a relation schema may satisfy a functional dependency even if the functional dependency does not hold on all legal instances
  - For example, a specific instance of *instructor* may, by chance, satisfy  $name \rightarrow ID$

# Functional Dependencies (Cont.)

- A functional dependency is **trivial** if it is satisfied by all instances of a relation

- Example:

- $ID, \underline{name} \rightarrow ID$  ✓
  - $\underline{name} \rightarrow name$

- In general,  $\alpha \rightarrow \beta$  is trivial if  $\beta \subseteq \alpha$

↳ present or future

$\beta$  is subset of  $\alpha$

on RHS is subset of LHS

# Functional Dependencies (Cont.)

PPD

- Functional dependencies are:

| StudentID | Semester | Lecture           | TA    |
|-----------|----------|-------------------|-------|
| 1234      | 6        | Numerical Methods | John  |
| 1221      | 4        | Numerical Methods | Smith |
| 1234      | 6        | Visual Computing  | Bob   |
| 1201      | 2        | Numerical Methods | Peter |
| 1201      | 2        | Physics II        | Simon |

- $StudentID \rightarrow Semester$
- $\{StudentID, Lecture\} \rightarrow TA$
- $\{StudentID, Lecture\} \rightarrow \{TA, Semester\}$

# Functional Dependencies (Cont.)

- Functional dependencies are:

| Employee ID | Employee Name | Department ID | Department Name |
|-------------|---------------|---------------|-----------------|
| 0001        | John Doe      | 1             | Human Resources |
| 0002        | Jane Doe      | 2             | Marketing       |
| 0003        | John Smith    | 1             | Human Resources |
| 0004        | Jane Goodall  | 3             | Sales           |

- $Employee\ ID \rightarrow Employee\ Name$
- $Employee\ ID \rightarrow Department\ ID$
- $Department\ ID \rightarrow Department\ Name$

# Closure of a Set of Functional Dependencies

- Given a set  $F$  of functional dependencies, there are certain other functional dependencies that are logically implied by  $F$ 
  - For example: If  $A \rightarrow B$  and  $B \rightarrow C$ , then we can infer that  $A \rightarrow C$
- The set of **all** functional dependencies logically implied by  $F$  is the **closure** of  $F$
- We denote the *closure* of  $F$  by  $F^+$
- $F^+$  is a superset of  $F$ 
  - $F = \{A \rightarrow B, B \rightarrow C\}$
  - $F^+ = \{A \rightarrow B, B \rightarrow C, A \rightarrow C\}$

1

6

.

2

0

a

@

s

a

i

b

l

a

b

s

e

r

s

s

y

c

s

h

t

a

e

t

m

z

c

,

o

k

n

o

c

r

e

t

p

h

t

a

s

n

n

d

6

S

t

S

h

u

E

u

d

d

i

t

a

i

r

o

r

n

s

s

h

a

a

# Module Summary

- Identified the features of good relational design
- Familiarized with the First Normal Form
- Introduced the notion of Functional Dependencies

NPTEL

# Instructor and TAs

PPD

| Name                          | Mail                                                                     | Mobile     |
|-------------------------------|--------------------------------------------------------------------------|------------|
| Partha Pratim Das, Instructor | <a href="mailto:ppd@cse.iitkgp.ernet.in">ppd@cse.iitkgp.ernet.in</a>     | 9830030880 |
| Srijoni Majumdar, TA          | <a href="mailto:majumdarsrijoni@gmail.com">majumdarsrijoni@gmail.com</a> | 9674474267 |
| Himadri B G S Bhuyan, TA      | <a href="mailto:himadribhuyan@gmail.com">himadribhuyan@gmail.com</a>     | 9438911655 |
| Gurunath Reddy M              | <a href="mailto:mgurunathreddy@gmail.com">mgurunathreddy@gmail.com</a>   | 9434137638 |

Slides used in this presentation are borrowed from <http://db-book.com/> with kind permission of the authors.

Edited and new slides are marked with “PPD”.



## Database Management Systems

# Module 17: Relational Database Design/2



**Partha Pratim Das**

*Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur*

[ppd@cse.iitkgp.ernet.in](mailto:ppd@cse.iitkgp.ernet.in)

Srijoni Majumdar  
Himadri B G S Bhuyan  
Gurunath Reddy M



---

**Database System Concepts, 6<sup>th</sup> Ed.**

©Silberschatz, Korth and Sudarshan  
[www.db-book.com](http://www.db-book.com)

# Module Recap

PPD

- Features of Good Relational Design
- Atomic Domains and First Normal Form
- Functional Dependencies

NPTEL

# Module Objectives

PPD

- To understand how a schema can be decomposed for a 'good' design using functional dependencies
- To introduce the theory of functional dependencies

NPTEL

# Module Outline

PPD

- Decomposition Using Functional Dependencies
- Functional Dependency Theory

NPTEL

- Decomposition Using Functional Dependencies
- Functional Dependency Theory

# DECOMPOSITION USING FUNCTIONAL DEPENDENCIES

# Boyce-Codd Normal Form

given

$$\begin{array}{c} \underline{R} \\ F = \{ fde \} \\ \Downarrow \\ \underline{F^+} \end{array}$$

$$\alpha \rightarrow \beta$$

where  $\alpha \subseteq R$  and  $\beta \subseteq R$ , at least one of the following holds:

- $\alpha \rightarrow \beta$  is trivial (i.e.,  $\beta \subseteq \alpha$ )
- $\alpha$  is a superkey for  $R$

$$RHS \subseteq LHS$$

Example schema not in BCNF:

$$\underline{R} = instr\_dept (ID, name, salary, \underline{dept\_name}, building, budget)$$

because  $dept\_name \rightarrow building, budget$  holds on  $instr\_dept$ , but  $dept\_name$  is not a superkey

$$\alpha \rightarrow \beta \quad \underline{F} \subseteq \alpha \text{ fails}$$

# Decomposing a Schema into BCNF

- Suppose we have a schema  $R$  and a non-trivial dependency  $\alpha \rightarrow \beta$  causes a violation of BCNF

We decompose  $R$  into:

$$\begin{aligned} R_1 &= (\alpha \cup \beta) \\ R_2 &= (R - (\beta - \alpha)) \end{aligned}$$

- In our example,

- $\alpha = \text{dept\_name}$
- $\beta = \text{building, budget}$

$\text{dept\_name} \rightarrow \text{building, budget}$

$\text{inst\_dept}$  is replaced by

- $(\alpha \cup \beta) = (\text{dept\_name}, \text{building, budget})$   $R_1$
- $\text{dept\_name} \rightarrow \text{building, budget}$

- $(R - (\beta - \alpha)) = (\text{ID, name, salary, dept\_name})$   $R_2$
- $\text{ID} \rightarrow \text{name, salary, dept\_name}$

$\beta - \alpha = (\text{building, budget})$

$R_1$  &  $R_2$  are in BCNF

~~Set a time only 2 decomposition~~

# BCNF and Dependency

## Preservation

- Constraints, including functional dependencies, are costly to check in practice unless they pertain to only one relation.
- If it is sufficient to test only those dependencies on each individual relation of a decomposition in order to ensure that all functional dependencies hold, then that decomposition is dependency preserving.
- Because it is not always possible to achieve both BCNF and dependency preservation, we consider a weaker normal form, known as third normal form.

# Third Normal Form

$\alpha \rightarrow \beta \text{ in } F^+$

- A relation schema  $R$  is in **third normal form (3NF)** if for all:

at least one of the following holds:

BCNF

- $\alpha \rightarrow \beta$  is trivial (i.e.,  $\beta \subseteq \alpha$ )
  - $\alpha$  is a superkey for  $R$
  - Each attribute  $A$  in  $\beta - \alpha$  is contained in a candidate key for  $R$
- (**NOTE:** each attribute may be in a different candidate key)
- If a relation is in BCNF it is in 3NF (since in BCNF one of the first two conditions above must hold)
  - Third condition is a minimal relaxation of BCNF to ensure dependency preservation (will see why later)

1NF  $\rightarrow$  atomic val  
2NF  $\rightarrow$  ?

3NF  $\Rightarrow$  BCNF +  
 $\alpha \in (B-1) \Leftrightarrow$   
 $R \text{ CNF} \Rightarrow$  trivial  
superkey

# Goals of Normalization

- Let  $R$  be a relation scheme with a set  $F$  of functional dependencies
- Decide whether a relation scheme  $R$  is in “good” form
- In the case that a relation scheme  $R$  is not in “good” form, decompose it into a set of relation scheme  $\{R_1, R_2, \dots, R_n\}$  such that
  - each relation scheme is in good form
  - the decomposition is a lossless-join decomposition
  - Preferably, the decomposition should be dependency preserving

all fds should hold  
on decomposed in new Relation

# How good is BCNF?

- There are database schemas in BCNF that do not seem to be sufficiently normalized
- Consider a relation

*inst\_info (ID, child\_name, phone)*

- where an instructor may have more than one phone and can have multiple children

| <i>ID</i> | <i>child_name</i> | <i>phone</i> |
|-----------|-------------------|--------------|
| 99999     | David             | 512-555-1234 |
| 99999     | David             | 512-555-4321 |
| 99999     | William           | 512-555-1234 |
| 99999     | Willian           | 512-555-4321 |

*inst\_info*

# How good is BCNF? (Cont.)

- There are no non-trivial functional dependencies and therefore the relation is in BCNF
- Insertion anomalies – i.e., if we add a phone 981-992-3443 to 99999,  
we need to add two tuples

(99999, David, 981-992-3443)  
(99999, William, 981-992-3443)

## How good is BCNF? (Cont.)

- Therefore, it is better to decompose *inst\_info* into:

*inst\_child*

| <i>ID</i> | <i>child_name</i> |
|-----------|-------------------|
| 99999     | David             |
| 00009     | David             |
| 99999     | William           |
| 99999     | Willian           |

*inst\_phone*

| <i>ID</i> | <i>phone</i> |
|-----------|--------------|
| 99999     | 512-555-1234 |
| 99999     | 512-555-4321 |
| 99999     | 512-555-1234 |
| 99999     | 512-555-4321 |

This suggests the need for higher normal forms, such as Fourth Normal Form (4NF)

*inst in syllabus*



PPD

- Decomposition Using Functional Dependencies
- **Functional Dependency Theory**

# FUNCTIONAL DEPENDENCY THEORY

# Functional- Dependency

## Theory

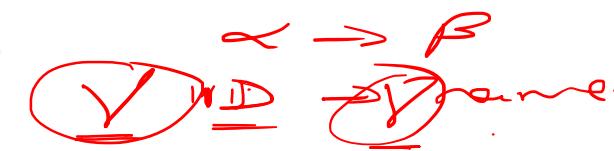
- We now consider the formal theory that tells us which functional dependencies are implied logically by a given set of functional dependencies
- We then develop algorithms to generate lossless decompositions into BCNF and 3NF
- We then develop algorithms to test if a decomposition is dependency-preserving

# Closure of a Set of Functional Dependencies

- Given a set  $F$  of functional dependencies, there are certain other functional dependencies that are logically implied by  $F$ 
  - For e.g.: If  $A \rightarrow B$  and  $B \rightarrow C$ , then we can infer that  $A \rightarrow C$
- The set of **all** functional dependencies logically implied by  $F$  is the **closure** of  $F$
- We denote the closure of  $F$  by  $F^+$

# Closure of a Set of Functional Dependencies

- We can find  $F^+$ , the closure of  $F$ , by repeatedly applying Armstrong's Axioms:
  - if  $\beta \subseteq \alpha$ , then  $\underline{\alpha \rightarrow \beta}$  **(reflexivity)**
  - if  $\alpha \rightarrow \beta$ , then  $\gamma \alpha \rightarrow \gamma \beta$  **(augmentation)**
  - if  $\alpha \rightarrow \beta$ , and  $\beta \rightarrow \gamma$ , then  $\alpha \rightarrow \gamma$  **(transitivity)**
- These rules are
  - **sound** (generate only functional dependencies that actually hold), and
  - **complete** (generate all functional dependencies that hold)



# Example

- $R = (A, B, C, G, H, I)$   
 $F = \{ A \rightarrow B$   
 $\quad \quad A \rightarrow C$   
 $\quad \quad CG \rightarrow H$   
 $\quad \quad CG \rightarrow I$   
 $\quad \quad B \rightarrow H \}$

$F^+ = \{ F, \text{ new } \}$

- Some members of  $F^+$

- $A \rightarrow H$  ✓
  - by transitivity from  $A \rightarrow B$  and  $B \rightarrow H$
- $AG \rightarrow I$  ✓
  - by augmenting  $A \rightarrow C$  with  $G$ , to get  $AG \rightarrow CG$  and then transitivity with  $CG \rightarrow I$
- $CG \rightarrow HI$ 
  - by augmenting  $CG \rightarrow I$  to infer  $CG \rightarrow CGI$ , and augmenting of  $CG \rightarrow H$  to infer  $CG \rightarrow HI$ , and then transitivity



# Procedure for Computing $F^+$

*(Closure of F)*

- To compute the closure of a set of functional dependencies  $F$ :

$$F^+ = F$$

repeat

for each functional dependency  $f$  in  $F^+$

apply reflexivity and augmentation rules on  $f$

add the resulting functional dependencies to  $F^+$

for each pair of functional dependencies  $f_1$  and  $f_2$  in  $F^+$

if  $f_1$  and  $f_2$  can be combined using transitivity

then add the resulting functional dependency to  $F^+$

until  $F^+$  does not change anyfurther

**NOTE:** We shall see an alternative procedure for this task later

# Closure of Functional Dependencie s (Cont.)

$F^+$   $\rightarrow G$  ~~subset~~

- Additional rules:

- If  $\alpha \rightarrow \beta$  holds and  $\alpha \rightarrow \gamma$  holds, then  $\alpha \rightarrow \beta\gamma$  holds (**union**)
- If  $\alpha \rightarrow \beta\gamma$  holds, then  $\alpha \rightarrow \beta$  holds and  $\alpha \rightarrow \gamma$  holds (**decomposition**)
- If  $\alpha \rightarrow \beta$  holds and  $\beta \rightarrow \delta$  holds, then  $\alpha \rightarrow \delta$  holds (**pseudotransitivity**)

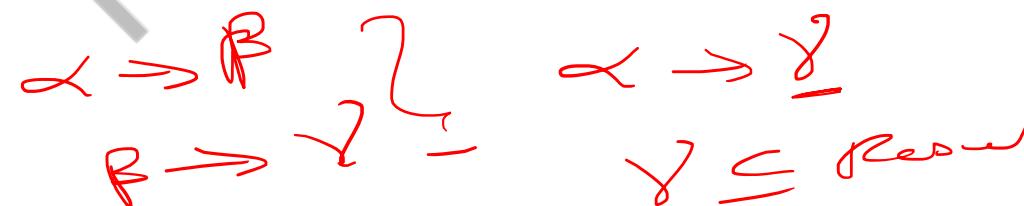
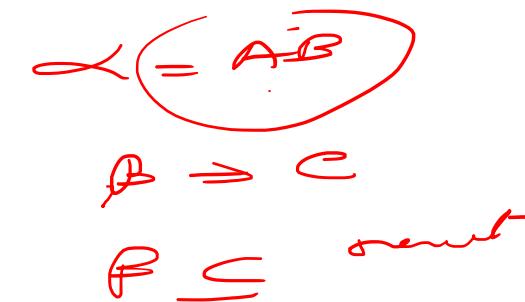
The above rules can be inferred from Armstrong's axioms

$$\begin{array}{ccc} \alpha & \rightarrow & \beta \\ \gamma\alpha & \rightarrow & \gamma\beta & \text{transitivity} \\ \gamma\beta & \rightarrow & \gamma\delta \\ \gamma\alpha & \rightarrow & \gamma\delta \end{array}$$

# Closure of Attribute Sets

- Given a set of attributes  $\alpha$ , define the **closure** of  $\alpha$  under  $F$  (denoted by  $\alpha^+$ ) as the set of attributes that are functionally determined by  $\alpha$  under  $F$
- Algorithm to compute  $\alpha^+$ , the closure of  $\alpha$  under  $F$

```
result := α ;
while (changes to result) do
 for each $\beta \rightarrow \gamma$ in F do
 begin
 if $\beta \subseteq result$ then $result := result \cup \gamma$
 end
```



# Example of Attribute Set Closure

- $R = (A, B, C, G, H, I)$

- $F = \{A \rightarrow B$

$$A \rightarrow C$$

$$\cancel{CG \rightarrow H}$$

$$\cancel{CG \rightarrow I}$$

$$\cancel{B \rightarrow H}\}$$

Ques:  $(AG)^+$

1.  $result = AG$

2.  $result = \cancel{ABC}G$  ( $A \rightarrow C$  and  $A \rightarrow B$ )

3.  $result = ABCGH$  ( $CG \rightarrow H$  and  $CG \subseteq AGBC$ )

4.  $result = ABCGHI$  ( $CG \rightarrow I$  and  $CG \subseteq AGBCH$ )

- Is  $AG$  a candidate key?

1. Is  $AG$  a superkey?

1. Does  $AG \rightarrow R$ ? == Is  $(AG)^+ \supseteq R$

2. Is any subset of  $AG$  a superkey?

1. Does  $A \rightarrow R$ ? == Is  $(A)^+ \supseteq R$

2. Does  $G \rightarrow R$ ? == Is  $(G)^+ \supseteq R$

Ans:

$$A \rightarrow R \subset C \supseteq G + T$$

# Uses of Attribute Closure

- There are several uses of the attribute closure algorithm:
  - Testing for superkey:
    - To test if  $\alpha$  is a superkey, we compute  $\alpha^+$ , and check if  $\alpha^+$  contains all attributes of  $R$ .
  - Testing functional dependencies
    - To check if a functional dependency  $\alpha \rightarrow \beta$  holds (or, in other words, is in  $F^+$ ), just check if  $\beta \subseteq \alpha^+$ .
    - That is, we compute  $\alpha^+$  by using attribute closure, and then check if it contains  $\beta$ .
    - Is a simple and cheap test, and very useful
  - Computing closure of  $F$ 
    - For each  $\gamma \subseteq R$ , we find the closure  $\gamma^+$ , and for each  $S \subseteq \gamma^+$ , we output a functional dependency  $\gamma \rightarrow S$ .

# Module Summary

- Discussed issues in ‘good’ design in the context of functional dependencies
- Introduced the theory of functional dependencies

NPTEL

# Instructor and TAs

PPD

| Name                          | Mail                                                                     | Mobile     |
|-------------------------------|--------------------------------------------------------------------------|------------|
| Partha Pratim Das, Instructor | <a href="mailto:ppd@cse.iitkgp.ernet.in">ppd@cse.iitkgp.ernet.in</a>     | 9830030880 |
| Srijoni Majumdar, TA          | <a href="mailto:majumdarsrijoni@gmail.com">majumdarsrijoni@gmail.com</a> | 9674474267 |
| Himadri B G S Bhuyan, TA      | <a href="mailto:himadribhuyan@gmail.com">himadribhuyan@gmail.com</a>     | 9438911655 |
| Gurunath Reddy M              | <a href="mailto:mgurunathreddy@gmail.com">mgurunathreddy@gmail.com</a>   | 9434137638 |

Slides used in this presentation are borrowed from <http://db-book.com/> with kind permission of the authors.

Edited and new slides are marked with “PPD”.

Canonical cover  
Lossless join  
dependency preserving joins

~~Work out~~

PPD

# Practice Problems on Functional Dependencies

## ■ Find Prime and Non Prime Attributes using Functional Dependencies:

1. R(ABCDEF) having FDs {AB→C, C→D, D→E, F→B, E→F}
2. R(ABCDEF) having FDs {AB → C, C → DE, E → F, C → B}
3. R(ABCDEFGHIJ) having FDs {AB → C, A → DE, B → F, F → GH, D → IJ}
4. R(ABDLPT) having FDs {B → PT, A → D, T → L}
5. R(ABCDEFGH) having FDs {E → G, AB → C, AC → B, AD → E, B → D, BC → A}
6. R(ABCDE) having FDs {A → BC, CD → E, B → D, E → A}
7. R(ABCDEH) having FDs {A → B, BC → D, E → C, D → A}

$\text{PA} : \{B, C\}$   
 $\text{Non PA} = \{D, E, F\}$

- **Prime Attributes** – Attribute set that belongs to any candidate key are called Prime Attributes
  - It is union of all the candidate key attribute: {CK1 ∪ CK2 ∪ CK3 ∪ .....}
  - If Prime attribute determined by other attribute set, then more than one candidate key is possible.
  - For example, If A is Candidate Key, and X→A, then, X is also Candidate Key .
- **Non Prime Attribute** – Attribute set does not belongs to any candidate key are called Non Prime Attributes

Source: <http://www.edugrabs.com/prime-and-non-prime-attributes/>

R(ABCDEF) having FDs {AB→C, C→D, D→E, F→B, E→F}

Candidate keys?

$$A^+ = A$$

$$AB^+ = ABCDEF = R \therefore \underline{AB} \text{ is CK}$$

~~$$AC^+ = ACDEFB = R \therefore \underline{AC} \text{ is CK}$$~~

$$AD^+ = ABCDEF = R \therefore \underline{AD} \text{ is CK}$$

$$AE^+ = ABCDEF = R \therefore \underline{AE} \text{ is CK}$$

~~$$AF^+ = AFBCDE = R \therefore \underline{AF} \text{ is CK}$$~~

Prime attributes: A B C D E F

Non Prime attribute : m? ↴

- **Algorithms for Functional Dependencies**
  - Lossless Join Decomposition
  - Dependency Preservation

# ALGORITHMS FOR FUNCTIONAL DEPENDENCIES

# Canonical Cover

- Sets of functional dependencies may have redundant dependencies that can be inferred from the others
  - For example:  $\underline{A \rightarrow C}$  is redundant in:  $\boxed{\{A \rightarrow B, B \rightarrow C, A \rightarrow C\}}$
  - Parts of a functional dependency may be redundant
    - E.g.: on RHS:  $\cancel{\{A \rightarrow B, B \rightarrow C, A \rightarrow CD\}}$  can be simplified to  $\{A \rightarrow B, B \rightarrow C, A \rightarrow D\}$
  - In the forward: (1)  $A \rightarrow CD \rightarrow A \rightarrow C$  and  $A \rightarrow D$  (2)  $A \rightarrow B, B \rightarrow C \rightarrow A \rightarrow C$
- In the reverse: (1)  $A \rightarrow B, B \rightarrow C \rightarrow A \rightarrow C$  (2)  $A \rightarrow C, A \rightarrow D \rightarrow A \rightarrow CD$
- E.g.: on LHS:  $\{A \rightarrow B, B \rightarrow C, AC \rightarrow D\}$  can be simplified to  $\{A \rightarrow B, B \rightarrow C, A \rightarrow D\}$ 
  - In the forward: (1)  $A \rightarrow B, B \rightarrow C \rightarrow A \rightarrow C \rightarrow A \rightarrow AC$  (2)  $A \rightarrow AC, AC \rightarrow D \rightarrow A \rightarrow D$
  - In the reverse:  $A \rightarrow D \rightarrow AC \rightarrow D$  *no - 86 marks*
- Intuitively, a canonical cover of F is a “minimal” set of functional dependencies equivalent to F, having no redundant dependencies or redundant parts of dependencies



# Canonical Cover

PPD

- A **canonical cover** for  $F$  is a set of dependencies  $F_c$  such that
  - $F$  logically implies all dependencies in  $F_c$ , and
  - $F_c$  logically implies all dependencies in  $F$ , and ✓
  - No functional dependency in  $F_c$  contains an **extraneous attribute**, and
  - Each left side of functional dependency in  $F_c$  is unique
- To compute a canonical cover for  $F$ :  
**repeat**
  - 1 Use the union rule to replace any dependencies in  $F$   
 $\alpha_1 \rightarrow \beta_1$  and  $\alpha_1 \rightarrow \beta_2$  with  $\alpha_1 \rightarrow \beta_1 \beta_2$  ✓
  - 2 Find a functional dependency  $\alpha \rightarrow \beta$  with an  
 extraneous attribute either in  $\alpha$  or in  $\beta$   
/\* Note: test for extraneous attributes done using  $F_c$ , not  $F^*$ \*/  
 If an extraneous attribute is found, delete it from  $\alpha \rightarrow \beta$  ✓**until**  $F$  does not change
- Note: Union rule may become applicable after some extraneous attributes have been deleted, so it has to be re-applied

- *Minimal Sets of Functional Dependencies*
- *Irreducible Set of Functional Dependencies*

# Canonical

PPD

## Cover ~~F<sub>c</sub>~~ RHS

- { $A \rightarrow B, B \rightarrow C, A \rightarrow CD$ }  $\Rightarrow$  { $A \rightarrow B, \cancel{B \rightarrow C}, \cancel{A \rightarrow D}$ }
  - (1)  $\cancel{A \rightarrow CD} \Rightarrow A \rightarrow C$  and  $A \rightarrow D$  (2)  $A \rightarrow B, \cancel{B \rightarrow C} \Rightarrow A \rightarrow C$
  - $\cancel{A+ = ABCD}$   $\cancel{\cancel{P}}$
  
- { $A \rightarrow B, B \rightarrow C, A \rightarrow D$ }  $\Rightarrow$  { $A \rightarrow B, B \rightarrow C, A \rightarrow CD$ }
  - $\cancel{A \rightarrow B, B \rightarrow C} \Rightarrow A \rightarrow C$
  - $A \rightarrow C, A \rightarrow D \Rightarrow A \rightarrow CD$
  - $\cancel{A+ = ABCD}$   $\cancel{\cancel{F}}$

$$F = F_c$$

# Canonical Cover: LHS

- $\{A \rightarrow B, B \rightarrow C, AC \rightarrow D\} \xrightarrow{\quad} \{A \rightarrow B, B \rightarrow C, A \rightarrow D\}$ 
  - $A \rightarrow B, B \rightarrow C \xrightarrow{\quad} A \rightarrow C \xrightarrow{\quad} A \rightarrow AC$
  - $A \rightarrow AC, AC \rightarrow D \xrightarrow{\quad} A \rightarrow D$
  - $A+ = ABCD$
- $\{A \rightarrow B, B \rightarrow C, A \rightarrow D\} \xrightarrow{\quad} \{A \rightarrow B, B \rightarrow C, AC \rightarrow D\}$ 
  - $A \rightarrow D \xrightarrow{\quad} AC \rightarrow D$
  - $AC+ = ABCD$

$\alpha \rightarrow \beta$

# Extraneous Attributes

- Consider a set  $F$  of functional dependencies and the functional dependency  $\alpha \rightarrow \beta$  in  $F$ .
  - Attribute  $A$  is **extraneous** in  $\alpha$  if  $A \in \alpha$  and  $F$  logically implies  $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$ .
  - Attribute  $A$  is **extraneous** in  $\beta$  if  $A \in \beta$  and the set of functional dependencies  $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$  logically implies  $F$ .
- Note: Implication in the opposite direction is trivial in each of the cases above, since a “stronger” functional dependency always implies a weaker one
- Example: Given  $F = \{A \rightarrow C, AB \rightarrow C\}$ 
  - $B$  is extraneous in  $AB \rightarrow C$  because  $\{A \rightarrow C, AB \rightarrow C\}$  logically implies  $A \rightarrow C$  (i.e. the result of dropping  $B$  from  $AB \rightarrow C$ ).
  - $A^+ = AC$  in  $\{A \rightarrow C, AB \rightarrow C\}$
- Example: Given  $F = \{A \rightarrow C, AB \rightarrow CD\}$ 
  - $C$  is extraneous in  $AB \rightarrow CD$  since  $AB \rightarrow C$  can be inferred even after deleting  $C$
  - $AB^+ = ABCD$  in  $\{A \rightarrow C, AB \rightarrow D\}$

# Testing if an Attribute is Extraneous

- Consider a set  $F$  of functional dependencies and the functional dependency  $\alpha \rightarrow \beta$  in  $F$ .
- To test if attribute  $A \in \alpha$  is extraneous in  $\alpha$ 
  1. Compute  $(\{\alpha\} - A)^+$  using the dependencies in  $F$
  2. Check that  $(\{\alpha\} - A)^+$  contains  $\beta$ ; if it does,  $A$  is extraneous in  $\alpha$
- To test if attribute  $A \in \beta$  is extraneous in  $\beta$ 
  1. Compute  $\alpha^+$  using only the dependencies in  $F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ ,  
 $LHS^+$
  2. Check that  $\alpha^+$  contains  $A$ ; if it does,  $A$  is extraneous in  $\beta$

# Computing a Canonical Cover

- $R = (A, B, C)$   $F = \{A \rightarrow BC, B \rightarrow C, A \rightarrow B, AB \rightarrow C\}$
- Combine  $A \rightarrow BC$  and  $A \rightarrow B$  into  $\underline{A \rightarrow BC}$ 
  - Set is now  $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$
- $A$  is extraneous in  $\underline{AB \rightarrow C}$ 
  - Check if the result of deleting  $A$  from  $\underline{AB \rightarrow C}$  is implied by the other dependencies
    - Yes: in fact,  $B \rightarrow C$  is already present!
  - Set is now  $\{A \rightarrow BC, B \rightarrow C\}$
- $C$  is extraneous in  $\underline{A \rightarrow BC}$ 
  - Check if  $A \rightarrow C$  is logically implied by  $A \rightarrow B$  and the other dependencies
    - Yes: using transitivity on  $A \rightarrow B$  and  $B \rightarrow C$ .
    - Can use attribute closure of  $A$  in more complex cases
- The canonical cover is:

$\alpha \rightarrow F'$   
 $\alpha \rightarrow F_2 \curvearrowright V$   
 Extraneous  
Repeat V

$A \rightarrow B$   
 $B \rightarrow C$   $\curvearrowright A \rightarrow C$

$A \rightarrow B$   
 $\underline{B \rightarrow C}$

$\hookrightarrow \Rightarrow \curvearrowright$

## Find the Minimal Cover or Irreducible Sets or Canonical Cover of a Set of Functional Dependencies:

1.  $AB \rightarrow CD$ ,  $BC \rightarrow D$

- Step 1: Use the union rule to replace any dependencies in  $F$

$\alpha_1 \rightarrow \beta_1$  and  $\alpha_1 \rightarrow \beta_2$  with  $\alpha_1 \rightarrow \beta_1 \beta_2$

So we can write  $\boxed{AB \rightarrow C, AB \rightarrow D, BC \rightarrow D}$

Step 2: find the redundancy step by step:

$$\overrightarrow{(AB)^+} = \{A, B, C, D\} \quad // \text{Now check for } AB \rightarrow C$$

$(AB)^+ = \{A, B, D\}$  without considering this FD :  $\underline{AB \rightarrow C}$

$$(A)^+ = \{A\}$$

$$(B)^+ = \{B\}$$

So  $AB \rightarrow C$  is essential not redundancy.[left side multiple attribute is also essential]

$$\overrightarrow{(AB)^+} = \{A, B, C, D\} \quad // \text{Now check for } \overline{AB \rightarrow D}$$

$\overline{(AB)^+} = \{A, B, C, D\}$  without considering this FD :  ~~$AB \rightarrow D$~~  [we got the same closure]

So  $AB \rightarrow D$  is redundancy.[SO delete this one]

$$\overrightarrow{(BC)^+} = \{B, C, D\} \quad // \text{Now check for } \overline{BC \rightarrow D}$$

$\overline{(BC)^+} = \{B, C\}$  without considering this FD :  $\underline{BC \rightarrow D}$

SO this FD :  $\underline{BC \rightarrow D}$  is essential.

So canonical cover set of the above set of FD is  $\overbrace{AB \rightarrow C, BC \rightarrow D}$

**Find the Minimal Cover or Irreducible Sets or Canonical Cover of a Set of Functional Dependencies:**

2.  $ABCD \rightarrow E$ ,  $E \rightarrow D$ ,  $AC \rightarrow D$ ,  $A \rightarrow B$

- Step 1: Use the union rule to replace any dependencies in  $F$

$\alpha_1 \rightarrow \beta_1$  and  $\alpha_1 \rightarrow \beta_2$  with  $\alpha_1 \rightarrow \beta_1 \beta_2$

We have nothing like this.

Step 2: find the redundancy step by step:

Now check for  $ABCD \rightarrow E$

$(ABCD)^+ = \{A, B, C, D, E\}$

$(ABCD)^+ = \{A, B, C, D\}$  without considering this FD :  $ABCD \rightarrow E$

$(A)^+ = \{A, B\}$        $=$

$(B)^+ = \{B\}$

$(C)^+ = \{C\}$

$(D)^+ = \{D\}$

So  $ABCD \rightarrow E$  is essential not redundancy.

Now check for  $E \rightarrow D$

$(E)^+ = \{E, D\}$

$(E)^+ = \{E\}$  without considering this FD:  $E \rightarrow D$ ,

So  $E \rightarrow D$  is essential.

**Find the Minimal Cover or Irreducible Sets or Canonical Cover of a Set of Functional Dependencies:**

$ABCD \rightarrow E, E \rightarrow D, AC \rightarrow D, A \rightarrow B$

Now check for  $AC \rightarrow D$ ,

$$\begin{aligned}(AC)^+ &= \{A, C, D\} // \text{USING } AC \rightarrow D \\ &= \{A, B, C, D\} // \text{USING } A \rightarrow B \\ &= \{A, B, C, D, E\} // \text{USING } ABCD \rightarrow E\end{aligned}$$

$(AC)^+ = \{A, B, C\}$  without considering this FD :  $AC \rightarrow D$

$(A)^+ = \{A, B\}$

$(C)^+ = \{C\}$

SO this FD :  $AC \rightarrow D$  is essential.[LEFT HAND ATTRIBUTE IS ALSO ESSENTIAL ]

Now check for  $A \rightarrow B$

$(A)^+ = \{A, B\}$

$(A)^+ = \{A\}$  without considering this FD:  $A \rightarrow B$

SO  $A \rightarrow B$  this FD is essential

So canonical cover set of the above set of FD is  $ABCD \rightarrow E, E \rightarrow D, AC \rightarrow D, A \rightarrow B$

# Equivalence of Sets of Functional Dependencies

- Let F & G are two functional dependency sets.
  - These two sets F & G are equivalent if  $F^+ = G^+$
  - Equivalence means that every functional dependency in F can be inferred from G, and every functional dependency in G can be inferred from F
- F and G are equal only if
  - F covers G: Means that all functional dependency of G are logically members of functional dependency set  $F \Rightarrow F \supseteq G$ .
  - G covers F: Means that all functional dependency of F are logically members of functional dependency set  $G \Rightarrow G \supseteq F$

| Condition  | CASES      |               |               |               |
|------------|------------|---------------|---------------|---------------|
|            | F Covers G | True          | True          | False         |
| G Covers F | True       | False         | True          | False         |
| Result     | $F=G$      | $F \supset G$ | $G \supset F$ | No Comparison |

# Practice Problems on Functional Dependencies

## ■ Check the Equivalence of a Pair of Sets of Functional Dependencies:

1. Consider the two sets F and G with their FDs as below :
  1.  $F : A \rightarrow C, AC \rightarrow D, E \rightarrow AD, E \rightarrow H$
  2.  $G : A \rightarrow CD, E \rightarrow AH$
2. Consider the two sets P and Q with their FDs as below :
  1.  $P : A \rightarrow B, AB \rightarrow C, D \rightarrow ACE$
  2.  $Q : A \rightarrow BC, D \rightarrow AE$

Source: <http://www.edugrabs.com/equivalence-of-sets-of-functional-dependencies-example/>

Check the Equivalence of a Pair of Sets of Functional Dependencies:

1. Consider the two sets F and G with their FDs as below :

- 1. F :  $A \rightarrow C, AC \rightarrow D, E \rightarrow AD, E \rightarrow H$
- 2. G:  $A \rightarrow CD, E \rightarrow AH$

- Step 1 : Find out the closure of all lefthand attributes of F functional dependency(FD) using the FD of G and vice versa.

| <u>F: <math>A \rightarrow C, AC \rightarrow D, E \rightarrow AD, E \rightarrow H</math></u>                      | <u>G: <math>A \rightarrow CD, E \rightarrow AH</math></u>                                                                 |
|------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| $(A)^+ = \{A, C, D\}$ //USING $A \rightarrow CD$ , OF FD G                                                       | $(A)^+ = \{A, C\}$ //USING $A \rightarrow C$ , OF FD F<br>$= \{A, C, D\}$ //USING $AC \rightarrow D$ , OF FD F            |
| $(AC)^+ = \{A, C, D\}$ //USING $A \rightarrow CD$ , OF FD G                                                      | $(E)^+ = \{E, A, D, H\}$ \\ USING $E \rightarrow AD, E \rightarrow H$<br>$= \{E, A, C, D, H\}$ \\ USING $A \rightarrow C$ |
| $(E)^+ = \{A, E, H\}$ // USING $E \rightarrow AH$ , OF FD G<br>$= \{A, C, D, E, H\}$ // USING $A \rightarrow CD$ | IF YOU WANT YOU CAN FIND FOR<br>$(AC)^+ = \{A, C, D\}$ //USING $AC \rightarrow D$ OF F                                    |
| $F \subseteq G$                                                                                                  | $G \subseteq F$                                                                                                           |

- $F \subseteq G$  and  $G \subseteq F$
- So  $F = G$
- So two sets of functional dependency is equivalent.

*F closure S*  
*S closure F*



PPD

- Algorithms for Functional Dependencies
- **Lossless Join Decomposition**
- Dependency Preservation

## LOSSLESS JOIN DECOMPOSITION

# Lossless-join Decomposition

- For the case of  $R = (R_1, R_2)$ , we require that for all possible relations  $r$  on schema  $R$
- $$r = \Pi_{R1}(r) \bowtie \Pi_{R2}(r)$$
- A decomposition of  $R$  into  $R_1$  and  $R_2$  is lossless join if at least one of the following dependencies is in  $F^+$ :
  - $R_1 \cap R_2 \rightarrow R_1$  ✓
  - $R_1 \cap R_2 \rightarrow R_2$  ✓
- The above functional dependencies are a sufficient condition for lossless join decomposition; the dependencies are a necessary condition only if all constraints are functional dependencies

To Identify whether a decomposition is lossy or lossless, it must satisfy the following conditions :

- $R_1 \cup R_2 = R$  ✓
- $R_1 \cap R_2 \neq \Phi$  and -
- $R_1 \cap R_2 \rightarrow R_1$  or  $R_1 \cap R_2 \rightarrow R_2$

# Example

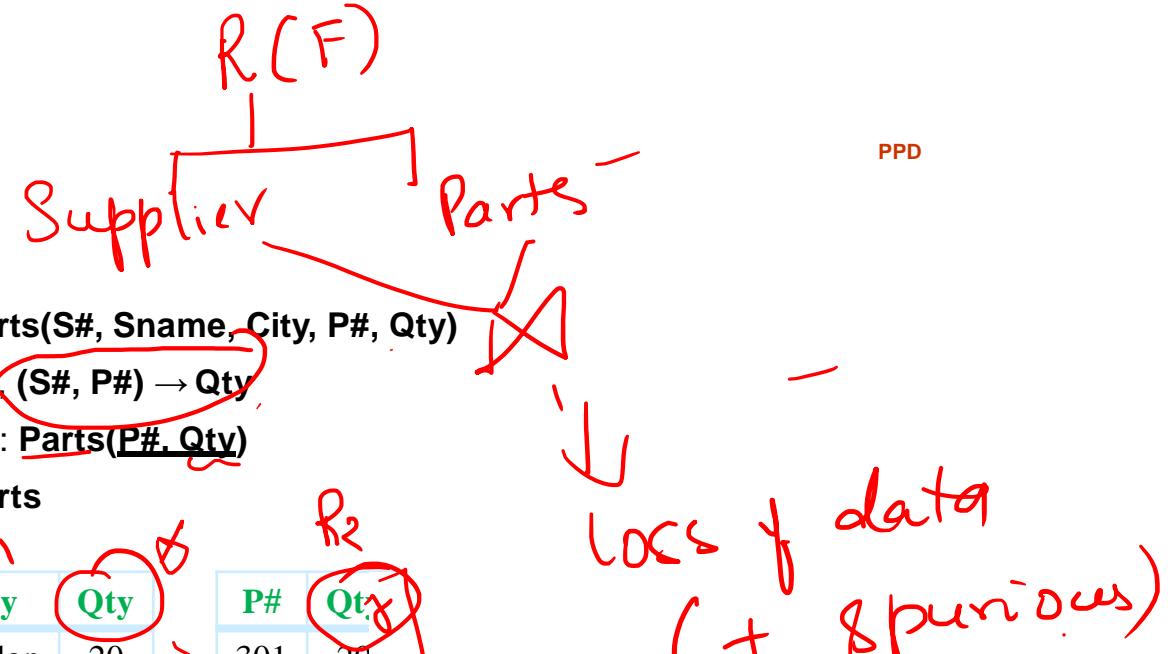
- Consider **Supplier\_Parts** schema: **Supplier\_Parts(S#, Sname, City, P#, Qty)**
- Having dependencies:  $S\# \rightarrow Sname$ ,  $S\# \rightarrow City$ ,  $(S\#, P\#) \rightarrow Qty$
- Decompose as: **Supplier(S#, Sname, City, Qty)**: **Parts(P#, Qty)**
- Take Natural Join to reconstruct: **Supplier  $\bowtie$  Parts**

| S# | Sname | City   | P#  | Qty |
|----|-------|--------|-----|-----|
| 3  | Smith | London | 301 | 20  |
| 5  | Nick  | NY     | 500 | 50  |
| 2  | Steve | Boston | 20  | 10  |
| 5  | Nick  | NY     | 400 | 40  |
| 5  | Nick  | NY     | 301 | 10  |

We get extra tuples! **Join is Lossy!**

- Common attribute **Qty** is not a superkey in **Supplier** or in **Parts**
- Does not preserve  $(S\#, P\#) \rightarrow Qty$

Source: <http://www.edugrabs.com/lossy-join-decomposition/>



| S# | Sname | City   | Qty | P#  | Qty |
|----|-------|--------|-----|-----|-----|
| 3  | Smith | London | 20  | 301 | 20  |
| 5  | Nick  | NY     | 50  | 500 | 50  |
| 2  | Steve | Boston | 10  | 20  | 10  |
| 5  | Nick  | NY     | 40  | 400 | 40  |
| 5  | Nick  | NY     | 10  | 301 | 10  |

| S# | Sname | City   | P#  | Qty |
|----|-------|--------|-----|-----|
| 3  | Smith | London | 301 | 20  |
| 5  | Nick  | NY     | 500 | 50  |
| 2  | Steve | Boston | 20  | 10  |
| 5  | Nick  | NY     | 400 | 40  |
| 5  | Nick  | NY     | 301 | 10  |
| 2  | Steve | Boston | 301 | 10  |

# Example

$R_1 \cap R_2 \rightarrow R_1 \text{ or } R_2$

PPD

3NF

- Consider **Supplier\_Parts** schema: **Supplier\_Parts(S#, Sname, City, P#, Qty)**
- Having dependencies:  $S\# \rightarrow Sname$ ,  $S\# \rightarrow City$ ,  $(S\#, P\#) \rightarrow Qty$
- Decompose as: **Supplier(S#, Sname, City): Parts(S#, P#, Qty)**
- Take Natural Join to reconstruct: **Supplier  $\bowtie$  Parts**

| S# | Sname | City   | P#  | Qty |
|----|-------|--------|-----|-----|
| 3  | Smith | London | 301 | 20  |
| 5  | Nick  | NY     | 500 | 50  |
| 2  | Steve | Boston | 20  | 10  |
| 5  | Nick  | NY     | 400 | 40  |
| 5  | Nick  | NY     | 301 | 10  |

| S# | Sname | City   |
|----|-------|--------|
| 3  | Smith | London |
| 5  | Nick  | NY     |
| 2  | Steve | Boston |
| 5  | Nick  | NY     |
| 5  | Nick  | NY     |

| S# | P#  | Qty |
|----|-----|-----|
| 3  | 301 | 20  |
| 5  | 500 | 50  |
| 2  | 20  | 10  |
| 5  | 400 | 40  |
| 5  | 301 | 10  |

✓

- We get back the original relation. **Join is Lossless.** *Decomposition*
- Common attribute  $S\#$  is a superkey in **Supplier**
- Preserves all dependencies

Source: <http://www.edugrabs.com/desirable-properties-of-decomposition/#lossless>

# Example

- $R = (A, B, C)$   
 $F = \{A \rightarrow B, B \rightarrow C\}$

- Can be decomposed in two different ways

- $R_1 = (A, B), R_2 = (B, C)$

- Lossless-join decomposition:

$$R_1 \cap R_2 = \{B\} \text{ and } B^+ = BC$$

- Dependency preserving ✓

- $R_1 = (A, B), \underline{R_2 = (A, C)}$

- Lossless-join decomposition: ✓

$$R_1 \cap R_2 = \{A\} \text{ and } A^+ = AB$$

- Not dependency preserving

(cannot check  $B \rightarrow C$  without computing  $R_1 \bowtie R_2$ )

→ lossless ?  
Dependency preserving  
decomposition ?

# Practice Problems on Lossless Join

PPD

$$R_1 \cap R_2 \rightarrow R_1 \text{ or } R_2$$

try  
==

## ■ Check if the decomposition of R into D is lossless:

1.  $R(ABC)$ :  $F = \{A \rightarrow B, A \rightarrow C\}$ .  $D = R_1(AB), R_2(BC)$
2.  $R(ABCDEF)$ :  $F = \{A \rightarrow B, B \rightarrow C, C \rightarrow D, E \rightarrow F\}$ .  $D = R_1(AB), R_2(BCD), R_3(DEF)$
3.  $R(ABCDEF)$ :  $F = \{A \rightarrow B, C \rightarrow DE, AC \rightarrow F\}$ .  $D = R_1(BE), R_2(ACDEF)$
4.  $R(ABCDEG)$ :  $F = \{AB \rightarrow C, AC \rightarrow B, AD \rightarrow E, B \rightarrow D, BC \rightarrow A, E \rightarrow G\}$ 
  1.  $D_1 = R_1(AB), R_2(BC), R_3(ABDE), R_4(EG)$
  2.  $D_2 = R_1(ABC), R_2(ACDE), R_3(ADG)$
5.  $R(ABCDEFGHIJ)$ :  $F = \{AB \rightarrow C, B \rightarrow F, D \rightarrow IJ, A \rightarrow DE, F \rightarrow GH\}$ 
  1.  $D_1 = R_1(ABC), R_2(ADE), R_3(BF), R_4(FGH), R_5(DIJ)$
  2.  $D_2 = R_1(ABCDE), R_2(BFGH), R_3(DIJ)$
  3.  $D_3 = R_1(ABCD), R_2(DE), R_3(BF), R_4(FGH), R_5(DIJ)$

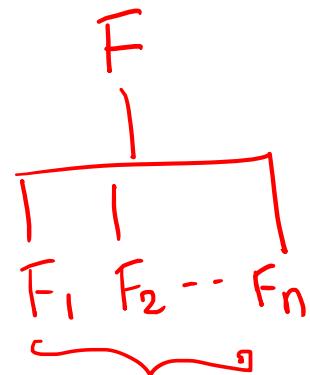
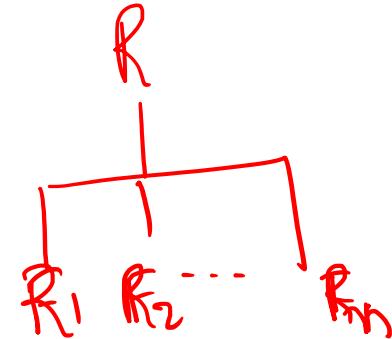
- Algorithms for Functional Dependencies
- Lossless Join Decomposition
- **Dependency Preservation**

# DEPENDENCY PRESERVATION



# Dependency Preservation

- Let  $F_i$  be the set of dependencies  $F^+$  that include only attributes in  $R_i$ 
  - A decomposition is **dependency preserving**, if  $(F_1 \cup F_2 \cup \dots \cup F_n)^+ = F^+$
  - If it is not, then checking updates for violation of functional dependencies may require computing joins, which is expensive



Let  $R$  be the original relational schema having FD set  $F$ . Let  $R_1$  and  $R_2$  having FD set  $F_1$  and  $F_2$  respectively, are the decomposed sub-relations of  $R$ . The decomposition of  $R$  is said to be preserving if

- $F_1 \cup F_2 \equiv F$  {Decomposition Preserving Dependency}
- $If F_1 \cup F_2 \subset F$  {Decomposition NOT Preserving Dependency} and
- $F_1 \cup F_2 \supset F$  {this is not possible}

# Testing for Dependency Preservation

- To check if a dependency  $\alpha \rightarrow \beta$  is preserved in a decomposition of  $R$  into  $R_1, R_2, \dots, R_n$  we apply the following test (with attribute closure done with respect to  $F$ )
  - $result = \alpha$
  - **while** (changes to  $result$ ) do
    - **for each**  $R_i$  in the decomposition
      - $t = (result \cap R_i)^+ \cap R_i$
      - $result = result \cup t$
  - If  $result$  contains all attributes in  $\beta$ , then the functional dependency  $\alpha \rightarrow \beta$  is preserved.
- We apply the test on all dependencies in  $F$  to check if a decomposition is dependency preserving
- This procedure takes polynomial time, instead of the exponential time required to compute  $F^+$  and  $(F_1 \cup F_2 \cup \dots \cup F_n)^+$

# Example

PPD

$$F = f_1 \dashv \dashv \dashv \dashv f_n$$

$$R_1 =$$

$$R_2 =$$

$$R_3 =$$

- $R(ABCDEF)$
  - $F = \{A \rightarrow BCD, A \rightarrow EF, BC \rightarrow AD, BC \rightarrow E, BC \rightarrow F, B \rightarrow F, D \rightarrow E\}$
  - $D = \{ABCD, BF, DE\}$
  - On projections:
- | $ABCD (R_1)$                               | $BF (R_2)$        | $DE (R_3)$        |
|--------------------------------------------|-------------------|-------------------|
| $A \rightarrow BCD$<br>$BC \rightarrow AD$ | $B \rightarrow F$ | $D \rightarrow E$ |

- Need to check for:  $A \rightarrow BCD, A \rightarrow EF, BC \rightarrow AD, BC \rightarrow E, BC \rightarrow F, B \rightarrow F, D \rightarrow E$
- $(BC)^+ / F_1 = ABCD$ .  $(ABCD)^+ / F_2 = ABCDF$ .  $(ABCDF)^+ / F_3 = ABCDEF$ . Preserves  $\underline{BC \rightarrow E}, \underline{BC \rightarrow F}$
- $(A)^+ / F_1 = ABCD$ .  $(ABCD)^+ / F_2 = ABCDF$ .  $(ABCDF)^+ / F_3 = ABCDEF$ . Preserves  $\underline{A \rightarrow EF}$

$\alpha \rightarrow \beta$   
 ↓ (start) find +  $\rightarrow \beta$

$$(BC)^+ / F_1 = (BCAD)^+ / F_2 = (BCADF)^+ / F_3 = \overbrace{BCADFE}^{\text{is dependency preserving decomposition}}$$

$$A^+ = ABCD \quad F_2 = (ABCD)^+ / F_3 = ABCDFG \quad \begin{cases} A \rightarrow EF \\ BF \rightarrow E \\ BC \rightarrow F \end{cases}$$

$\therefore$  is dependency preserving decomposition.

# Practice Problems on Dependency Preservation

PPD

- Check whether the decomposition of R into D is preserving dependency:

- 1. R(ABCDEF): F = {A→BCD, A→EF, BC→AD, BC→E, BC→F, B→F, D→E}. D = {ABCD, BF, DE} ✓
- ✓ 2. R(ABCD): F = {A → B, B → C, C → D, ~~D → A~~}. D = {AB, BC, CD}
- ✓ 3. R(ABCDEF): F = {AB → CD, C → D, D → E, E → F}. D = {AB, CDE, EF}

$$\underline{R_1(AB)}$$

$$A \rightarrow B$$

$$\underline{R_2(BC)}$$

$$B \rightarrow C$$

$$\underline{R_3(CD)}$$

$$C \rightarrow D$$

$$\text{Find } \overline{D \rightarrow A}$$

$$\underline{A^+ | F_1} = (AB)^+ | F_2 = (ABC)^+ | F_3 = \underline{\underline{ABCD}}$$

Source: <http://www.edugrabs.com/question-on-dependency-preserving-decomposition/>

∴ It is not dependency preserving decomposition

~~$$D \rightarrow A$$~~

$$\begin{array}{c} A \rightarrow D \\ + D \rightarrow A \\ = \end{array}$$



# Chapter 14: Transactions

Database System Concepts, 6<sup>th</sup> Ed.

©Silberschatz, Korth and Sudarshan

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



# Chapter 14: Transactions

- Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability ~~//~~
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability.



# Transactions

A series of one or more SQL statements that are logically related are termed as a Transaction.

E.g:

1. transaction to transfer \$50 from account A to account B:

1. ~~read(A)~~ ✓ *fetches*
  2. ~~A := A - 50~~
  3. ~~write(A)~~ → *update*
  4. ~~read(B)~~ ✓ *fetches*
  5. ~~B := B + 50~~
  6. ~~write(B)~~ - *update B*
- synonym for  
2nd Statement  
to  
over  
study purpose*

Sgj



- A **transaction** is a **unit** of program execution that accesses and possibly updates various data items. Usually, a transaction is initiated by a user program written in a high-level data-manipulation language
- A transaction is delimited by statements (or function calls) of the form **begin transaction** and **end transaction**. The transaction consists of all operations executed between the begin transaction and end transaction.
- This collection of steps must appear to the user as a single, indivisible unit. This “all-or-none” property is referred to as **atomicity**.



- the database system must take special actions to ensure that transactions operate properly without interference from concurrently executing database statements. This property is referred to as **isolation**.
- Even if the system ensures correct execution of a transaction, this serves little purpose if the system subsequently crashes and, as a result, the system “forgets” about the transaction. Thus, a transaction’s actions must persist across crashes. This property is referred to as **durability**.
- if a transaction is run atomically in isolation starting from a consistent database, the database must again be consistent at the end of the transaction. This property is referred to as **consistency**.

$$\text{Transfer } A \Rightarrow B = \sum \frac{(A \rightarrow B)}{+}$$



# ACID Properties

b c t

we require that the database system maintain the following properties of the transactions:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are. *single unit*
- **Consistency.** Execution of a transaction in **isolation** preserves the consistency of the database.
- **Isolation.** Although multiple transactions may **execute concurrently**, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions  $T_i$  and  $T_j$ , it appears to  $T_i$  that either  $T_j$  finished execution before  $T_i$  started, or  $T_j$  started execution after  $T_i$  finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the **database persist**, even if there are system failures.



# Example of Fund Transfer



- Transaction to transfer \$50 from account A to account B:

1. **read(A)**
2.  $A := A - 50$
3. **write(A)**
4. **read(B)**
5.  $B := B + 50$
6. **write(B)**

Explain properties with an example

self study

- **Atomicity requirement**

- if the transaction fails after step 3 and before step 6, money will be “lost” leading to an inconsistent database state

▶ Failure could be due to software(R.Error in Query) or hardware(S/W crash)

- the system should ensure that updates of a partially executed transaction are not reflected in the database

- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the \$50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.



# Example of Fund Transfer (Cont.)

□ Transaction to transfer \$50 from account A to account B:

1. **read(A)**
2.  $A := A - 50$
3. **write(A)**
4. **read(B)**
5.  $B := B + 50$
6. **write(B)**

□ **Consistency requirement** in above example:

- the sum of A and B is unchanged by the execution of the transaction

□ In general, consistency requirements include

- Explicitly specified integrity constraints such as primary keys and foreign keys
- **Implicit integrity constraints**
  - e.g. sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
- A transaction must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully the database must be consistent
  - Erroneous transaction logic can lead to inconsistency



# Example of Fund Transfer (Cont.)

- **Isolation requirement** — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum  $A + B$  will be less than it should be).

T1

1. **read(A)**
2.  $A := A - 50$
3. **write(A)**
4. **read(B)**
5.  $B := B + 50$
6. **write(B)**

T2

**read(A), read(B), print(A+B)**

- Isolation can be ensured trivially by running transactions **serially**
  - that is, one after the other.
- However, executing **multiple transactions concurrently has significant benefits**, as we will see later.



# Storage Structure

- To understand how to ensure the atomicity and durability properties of a transaction, we must gain a better understanding of how the various data items in the database may be stored and accessed.
- **Volatile storage:** Information residing in volatile storage does not usually survive system crashes. Examples: **main memory** and **cache memory**.
- **Nonvolatile storage:** Information residing in nonvolatile storage survives system crashes. Ex: **secondary storage** devices such as magnetic disk and flash storage, used for online storage, and tertiary storage devices such as optical media, and magnetic tapes, used for archival storage.
- **Stable storage:** Information residing in stable storage is *never lost*.  
*Extremely unlikely to lose data- replicate information*



- For a transaction to be durable, its changes need to be written to stable storage.
- Similarly, for a transaction to be atomic, log records need to be written to stable storage before any changes are made to the database on disk.
- Clearly, the degree to which a system ensures durability and atomicity depends on how stable its implementation of stable storage really is.



# Transaction Atomicity and Durability

Terminology

- A transaction may not always complete its execution successfully. Such a transaction is termed as aborted. *Fails*
- Once the changes caused by an aborted transaction have been undone, we say that the transaction has been rolled back.
- It is part of the responsibility of the recovery scheme to manage transaction aborts. This is done typically by maintaining a log.
- Maintaining a log provides the possibility of redoing a modification to ensure atomicity and durability as well as the possibility of undoing a modification to ensure atomicity in case of a failure during transaction execution.
- A transaction that completes its execution successfully is said to be committed. The only way to undo the effects of a committed transaction is to execute a **compensating transaction**.



# Transaction State

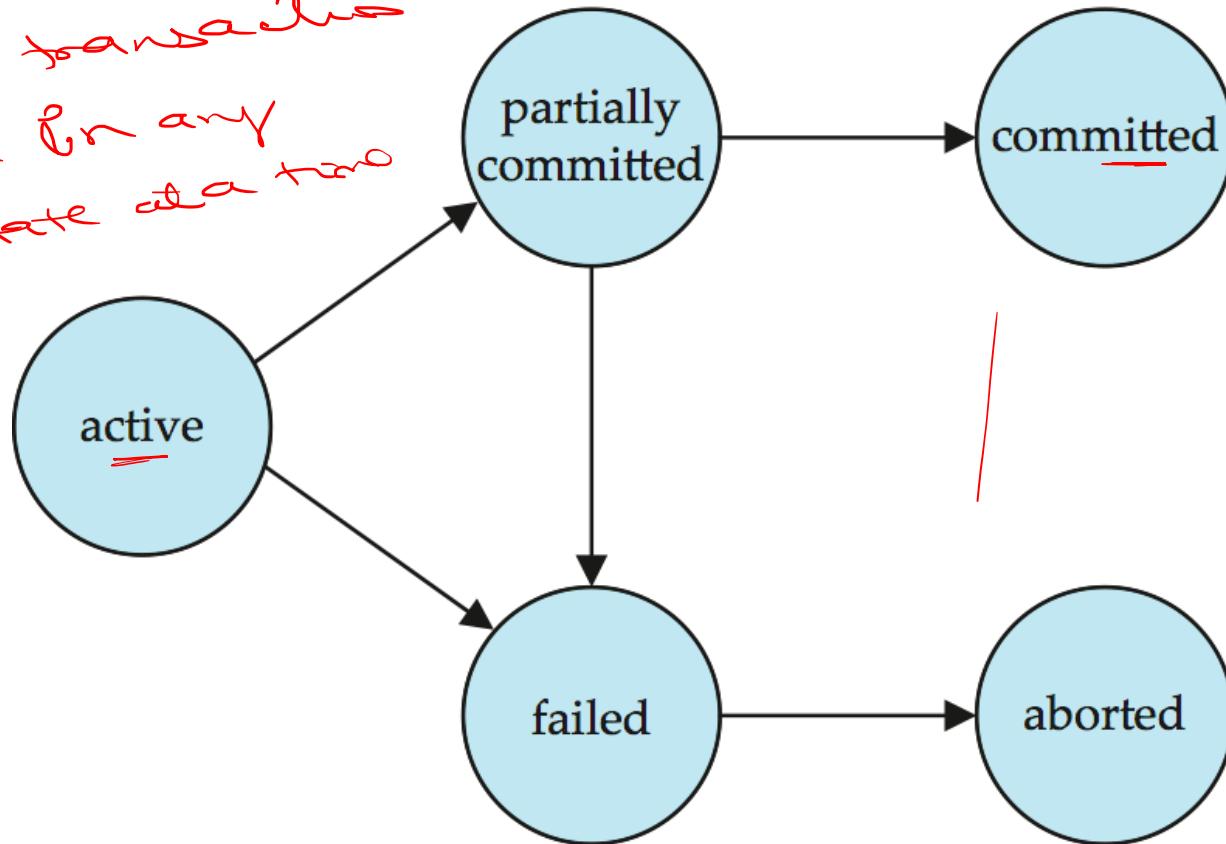
==

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed** -- after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction.  
Two options after it has been aborted:
  - restart the transaction
    - ▶ can be done only if no internal logical error
  - kill the transaction
- **Committed** – after successful completion.



# Transaction model

~~single transaction~~  
single transaction  
can be in any  
one state at a time





# Transaction State (Cont.)

- The system has two options if the transaction is failed:
- It can **restart** the transaction, but only if the transaction was aborted as a result of some hardware or software error that was not created through the internal logic of the transaction.
- A restarted transaction is considered to be a new transaction.
- It can **kill the transaction**. It usually does so because of some internal logical error that can be corrected only by rewriting the application program, or because the input was bad, or because the desired data were not found in the database.



# Transaction Concept

## COMMIT and ROLLBACK commands

- A transaction begins with the first executable SQL statement after a commit, rollback or connection made to the database.
- A transaction can be closed by using a commit or a rollback statement.
- **COMMIT** ends the current transaction and makes permanent changes made during the transaction
- **ROLLBACK** ends the transaction but undoes any changes made during the transaction
- Syntax: COMMIT;      =====      ROLLBACK;      =====





## Example for usage of **Commit, Rollback and SAVEPOINT**

- Withdraw an amount 2,000 and deposit 10,000 for all the accounts. If the sum of balance of all accounts exceeds 2,00,000 then undo the deposit just made.

```
Declare
 Total_bal numeric;
Begin
 Update account set balance=balance-2000;
 Savepoint deposit;
 Update account set balance=balance+10000;
 Select sum(balance) into total_bal from account;
 If total_bal > 200000 then
 Rollback to savepoint deposit;
 End if;
 Commit;
End;
```

Annotations:

- A red arrow points from the word "Savepoint" to the label "undo" on the right.
- A red circle highlights the word "Commit".
- A red bracket groups the "Savepoint deposit;" line and the "Rollback to savepoint deposit;" line, with a handwritten note "undo" next to it.

- SAVEPOINT** marks and saves the current point in the processing of a transaction. When a savepoint is used with a **ROLLBACK** statement, parts of transaction can be undone.



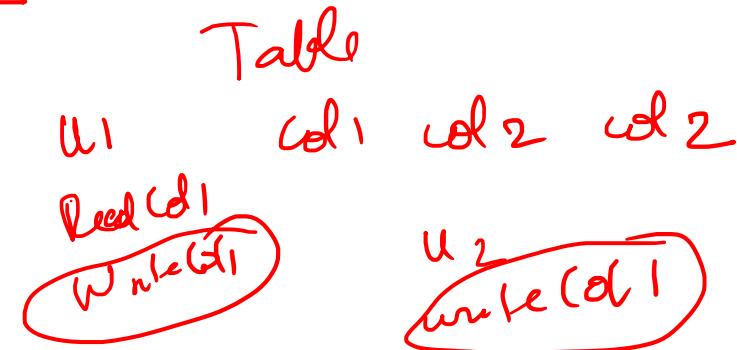
# Transaction isolation

- Ensuring consistency in spite of concurrent execution of transactions requires extra work; it is far easier to insist that transactions run **serially—that is, one at a time, each** starting only after the previous one has completed.
- Multiple transactions are allowed to run concurrently in the system.  
**Advantages are:**
  - **increased processor and disk utilization**, leading to better transaction *throughput*
    - ▶ E.g. one transaction can be using the CPU while another is reading from or writing to the disk
  - **reduced average response time/waiting time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation
  - that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the **consistency of the database**.



# Why Concurrency Control is needed

- Problems occur when concurrent transactions execute in an uncontrolled manner:
  - The Lost Update ✓
  - The Temporary Update (or Dirty Read)
  - The Incorrect Summary ✓
- Unrepeatable Read: ✓
  - ▶ A transaction T1 may read a given value. If another transaction later updates that value and T1 reads that value again, then T1 will see a different value.



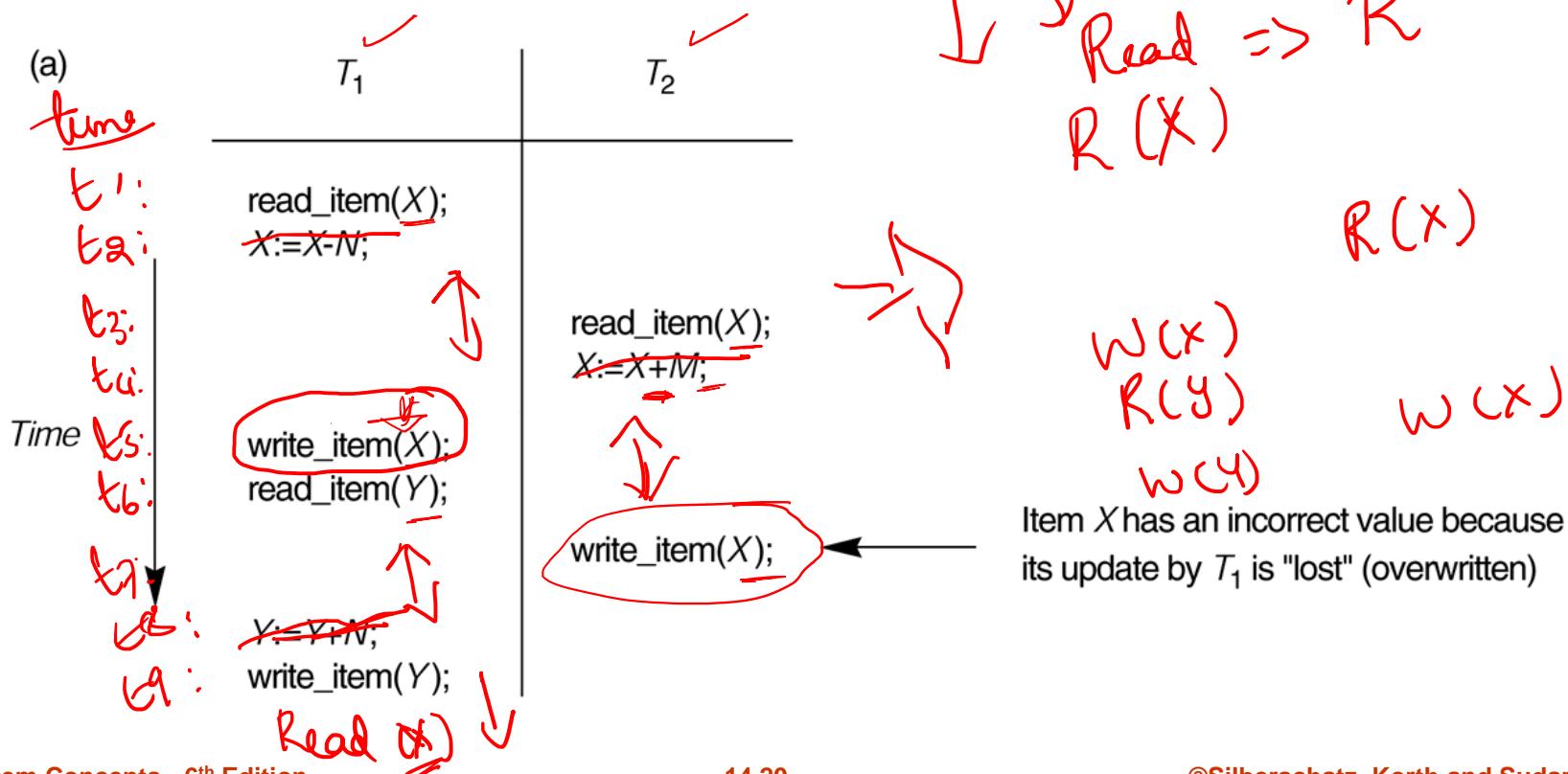


# The Lost Update Problem

data overwritten

- This occurs when **two** transactions that access the **same** database items have their operations interleaved in a way that makes the value of some database item incorrect.

*transaction isolation*



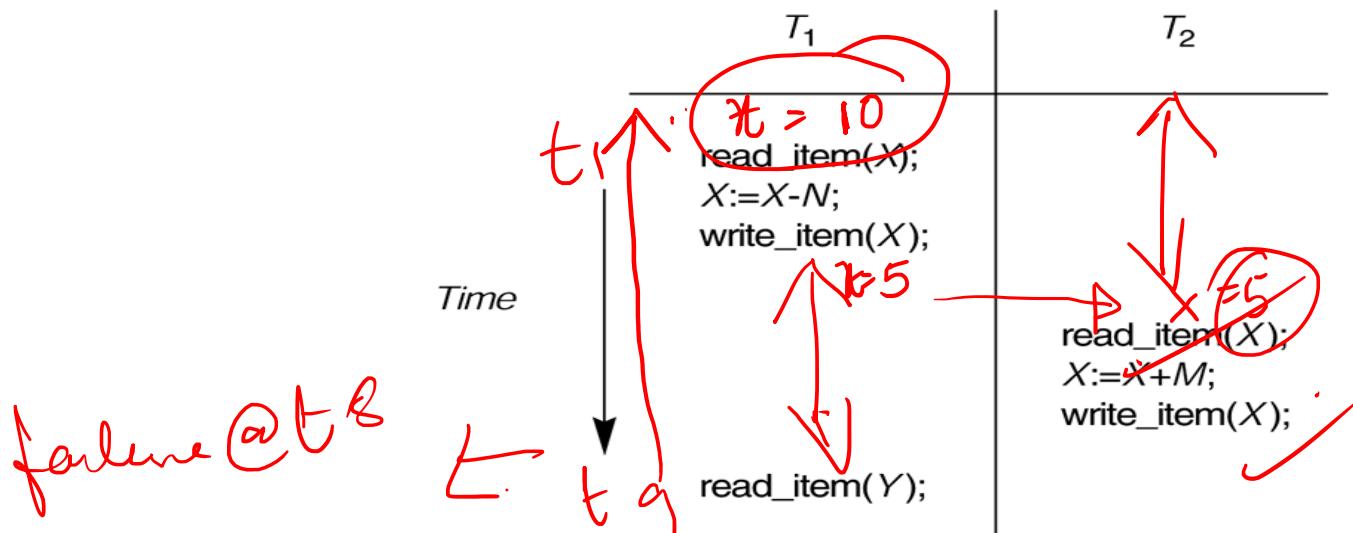


# The Temporary Update Problem (Dirty Read) ✓

- This occurs when one transaction updates a database item and then the transaction fails for some reason. *Rollback.*
- The updated item is accessed by another transaction before it is changed back to its original value.

=

(b)



Transaction  $T_1$  fails and must change the value of  $X$  back to its old value; meanwhile  $T_2$  has read the "temporary" incorrect value of  $X$ .



# Incorrect Summary Problem

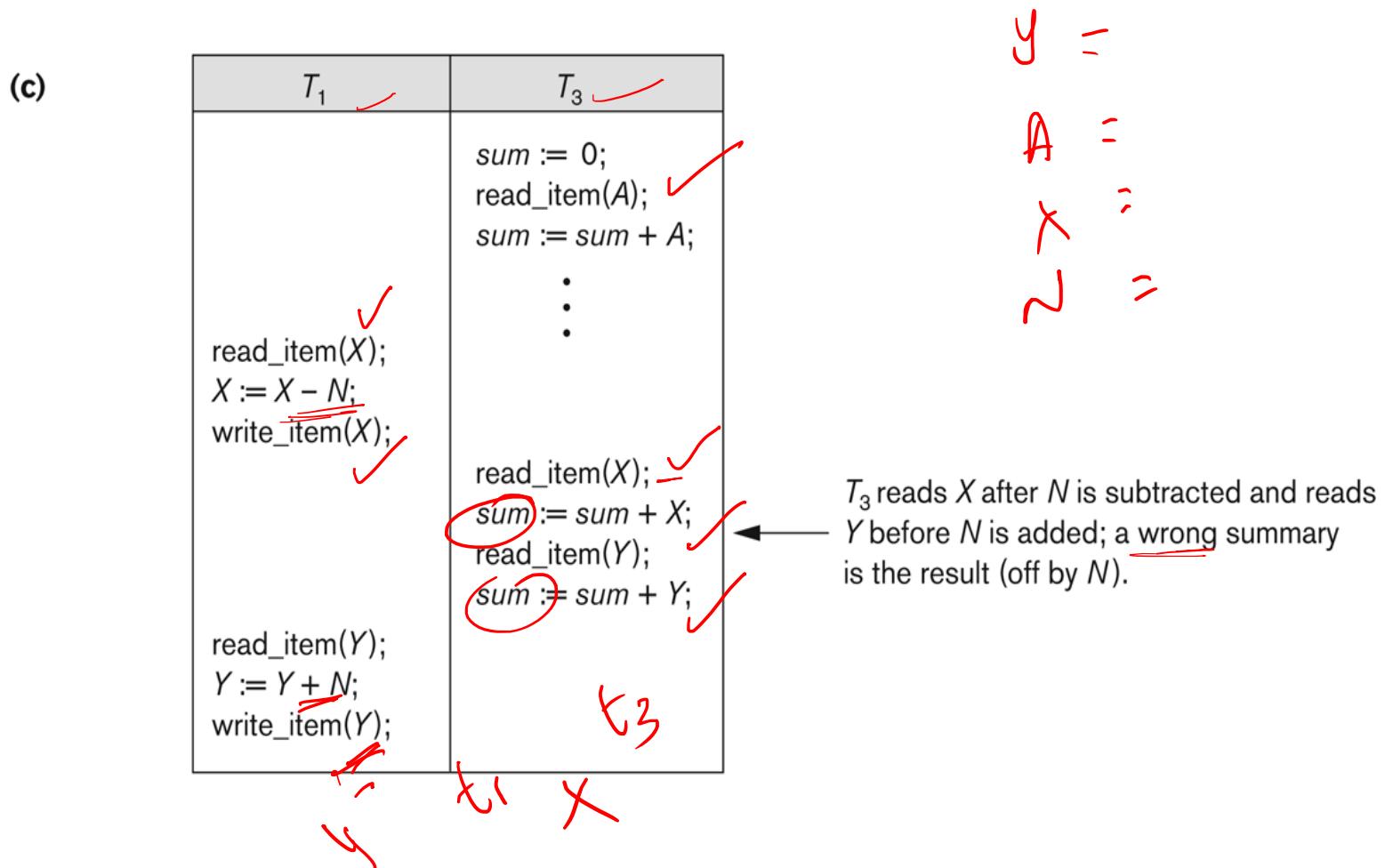
- If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.



# Incorrect Summary Problem

**Figure 17.3**

Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.





# Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - by default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement



# Schedules

$T_i \rightarrow n$  operation

- A **schedule**  $S$  of  $n$  transactions  $T_1, T_2, \dots, T_n$  is an ordering of **all** the operations in these transactions subject to the constraint that:
  - for each transaction  $T_i$ , the operations of  $T_i$  in  $S$  must appear in the **same order** as they do in  $T_i$ .
    - ▶ Note, however, that operations from other transactions  $T_j$  can be interleaved with the operations of  $T_i$  in  $S$ .

- Example: Given

- $T_1 = R_1(Q) W_1(Q)$  &  $T_2 = R_2(Q) W_2(Q)$

- a schedule:

~~$R_1(Q) R_2(Q) W_1(Q) W_2(Q)$~~

- not a schedule:

$W_1(Q) R_1(Q) R_2(Q) W_2(Q)$

$t_1 \rightarrow t_n$

$R_2(X)$   
 $W_2(B)$

Read =  $R_1$   
Write =  $W_1$

~~Conflict~~  
View



9/11/2020

# Schedule 1

- Let  $T_1$  transfer \$50 from  $A$  to  $B$ , and  $T_2$  transfer 10% of the balance from  $A$  to  $B$ .
- A **serial schedule** in which  $T_1$  is followed by  $T_2$ :

2 transactions  
is one schedule

no interleaving

| $T_1$                                                                           | $T_2$                                                                                               |
|---------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------|
| <pre>read (A) A := A - 50 write (A) read (B) B := B + 50 write (B) commit</pre> | <pre>read (A) temp := A * 0.1 A := A - temp write (A) read (B) B := B + temp write (B) commit</pre> |



# Schedule 2

- A serial schedule where  $T_2$  is followed by  $\underline{\underline{T_1}}$

| $T_1$                                                                                      | $T_2$                                                                                                               |
|--------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------|
| read (A)<br>$A := A - 50$<br>write (A)<br>read (B)<br>$B := B + 50$<br>write (B)<br>commit | read (A)<br>$temp := A * 0.1$<br>$A := A - temp$<br>write (A)<br>read (B)<br>$B := B + temp$<br>write (B)<br>commit |



# Schedule 3

- Let  $T_1$  and  $T_2$  be the transactions defined previously. The following schedule is not a serial schedule (It is CONCURRENT Schedule),
- but it is equivalent to Schedule 1.

|  | $T_1$                                                    | $T_2$                                                                 |
|--|----------------------------------------------------------|-----------------------------------------------------------------------|
|  | read ( $A$ )<br>$A := A - 50$<br>write ( $A$ )           | read ( $A$ )<br>$temp := A * 0.1$<br>$A := A - temp$<br>write ( $A$ ) |
|  | read ( $B$ )<br>$B := B + 50$<br>write ( $B$ )<br>commit | read ( $B$ )<br>$B := B + temp$<br>write ( $B$ )<br>commit            |

In Schedules 1, 2 and 3, the sum  $A + B$  is preserved.



# Schedule 4

- The following concurrent schedule does not preserve the value of  $(A + B)$ . It has Lost Update problems.

| $T_1$                                                                     | $T_2$                                                                                 |
|---------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| read ( $A$ )<br>$A := A - 50$                                             | read ( $A$ )<br>$temp := A * 0.1$<br>$A := A - temp$<br>write ( $A$ )<br>read ( $B$ ) |
| write ( $A$ )<br>read ( $B$ )<br>$B := B + 50$<br>write ( $B$ )<br>commit | $B := B + temp$<br>write ( $B$ )<br>commit                                            |

Lost update

Does not sum up.



- We can ensure consistency of the database under concurrent execution by making sure that any schedule that is executed has the same effect as a schedule that could have occurred without any concurrent execution.
- That is, the schedule should, in some sense, be equivalent to a serial schedule. Such schedules are called serializable schedules.
- Serial schedules are serializable, but if steps of multiple transactions are interleaved, it is harder to determine whether a schedule is serializable.
- Since transactions are programs, it is difficult to determine exactly what operations a transaction performs and how operations of various transactions interact.
- For this reason, we shall not consider the various types of operations that a transaction can perform on a data item, but instead consider only two operations: **read** and **write**.

A: A - R X



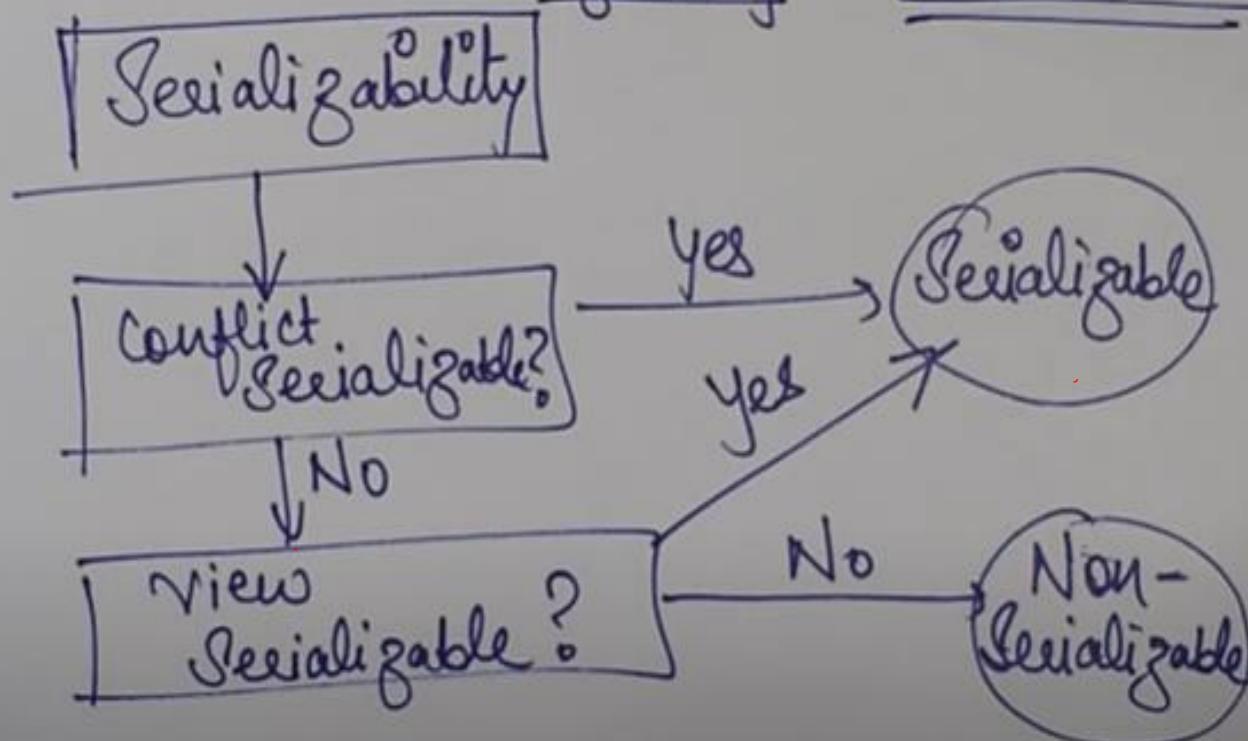
(X)

# Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
  - Thus serial execution of a set of transactions preserves database consistency.
  - A concurrent schedule is serializable if it is equivalent to a serial schedule.
- (Handwritten notes: T<sub>1</sub>, T<sub>2</sub>, op<sub>1</sub>, op<sub>2</sub>, op<sub>3</sub>, op<sub>4</sub>)*
- (Handwritten notes: 1. conflict serializability, 2. view serializability)*



## Serializability : A Broader View





# *Simplified view of transactions*

- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.



- In this section, we discuss different forms of schedule equivalence, but focus on a particular form called **conflict serializability**.
- Let us consider a schedule S in which there are two consecutive instructions, I and J, of transactions T<sub>i</sub> and T<sub>j</sub>, respectively ( $i \neq j$ ).
- there are four cases that we need to consider
- $I = \text{read}(Q)$ ,  $J = \text{read}(Q)$ . The order of  $I$  and  $J$  does not matter, since the same value of  $Q$  is read by  $T_i$  and  $T_j$ , regardless of the order.
- $I = \text{read}(Q)$ ,  $J = \text{write}(Q)$ . If  $I$  comes before  $J$ , then  $T_i$  does not read the value of  $Q$  that is written by  $T_j$  in instruction  $J$ . If  $J$  comes before  $I$ , then  $T_i$  reads the value of  $Q$  that is written by  $T_j$ . Thus, the order of  $I$  and  $J$  matters.



- $I = \text{write}(Q)$ ,  $J = \text{read}(Q)$ . The order of  $I$  and  $J$  matters for reasons similar to those of the previous case.
- $I = \text{write}(Q)$ ,  $J = \text{write}(Q)$ . Since both instructions are write operations, the order of these instructions does not affect either  $T_i$  or  $T_j$ .
- However, the value obtained by the **next read(Q)** instruction of  $S$  is affected, since the result of only the latter of the two write instructions is preserved in the database. If there is no other write( $Q$ ) instruction after  $I$  and  $J$  in  $S$ , then the order of  $I$  and  $J$  directly affects the final value of  $Q$  in the database state that results from schedule  $S$ .



- We say that  $I$  and  $J$  **conflict** if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.



# Conflict Serializability

## Conflicting Instructions

- Instructions  $I_i$  and  $I_j$  of transactions  $T_i$  and  $T_j$  respectively, **conflict if and only if** there exists some item  $Q$  accessed by both  $I_i$  and  $I_j$ , and at least one of these instructions wrote  $Q$ .

- R 
- $I_i = \text{read}(Q), I_j = \text{read}(Q)$ .  $I_i$  and  $I_j$  don't conflict.
  - $I_i = \text{read}(Q), I_j = \text{write}(Q)$ . They conflict.
  - $I_i = \text{write}(Q), I_j = \text{read}(Q)$ . They conflict
  - $I_i = \text{write}(Q), I_j = \text{write}(Q)$ . They conflict

- Intuitively, a conflict between  $I_i$  and  $I_j$  forces a (logical) **temporal order between them**.
  - If  $I_i$  and  $I_j$  are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.



# Conflict Serializability

*concurrent schedule*

- If a schedule  $\underline{S}$  can be transformed into a schedule  $\underline{S'}$  by a series of swaps of non-conflicting instructions, we say that  $S$  and  $S'$  are **conflict equivalent**.

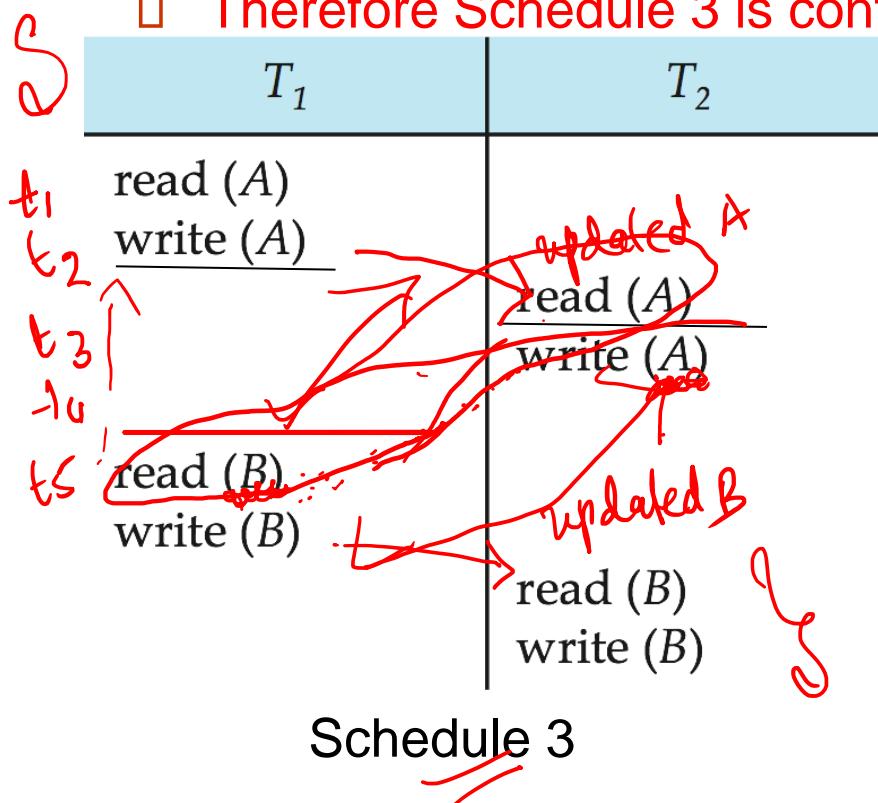
- We say that a **schedule  $S$**  is **conflict Serializable** if it is **conflict equivalent to a serial schedule**





# Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6, a serial schedule where  $T_2$  follows  $T_1$ , by series of swaps of non-conflicting instructions.
- Therefore Schedule 3 is conflict serializable.



*Result / Solution*

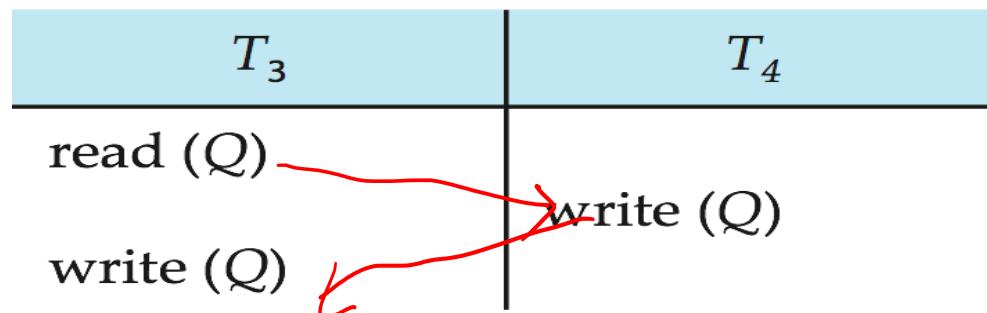
|       | $T_1$            | $T_2$     |
|-------|------------------|-----------|
| $t_1$ | read (A)         |           |
| $t_2$ | <u>write (A)</u> |           |
| $t_3$ |                  | read (B)  |
| $t_4$ |                  | write (B) |
|       |                  | read (A)  |
|       |                  | write (A) |
|       |                  | read (B)  |
|       |                  | write (B) |

*Schedule 6*



# Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:

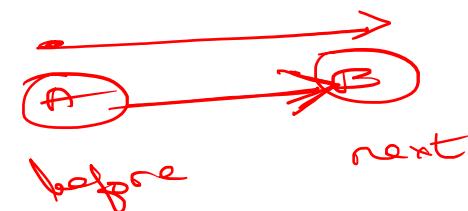


- We are **unable to swap instructions** in the above schedule to obtain either the serial schedule  $\langle T_3, T_4 \rangle$ , or the serial schedule  $\langle T_4, T_3 \rangle$ .



# Check Conflict Serializable, using Precedence Graph

- We now present a simple and efficient method for determining conflict serializability of a schedule. Consider a schedule  $S$ . We construct a directed graph, called a precedence graph, from  $S$ .
- This graph consists of a pair  $G = (V, E)$ , where  $V$  is a set of vertices and  $E$  is a set of edges. The set of vertices consists of all the transactions participating in the schedule. The set of edges consists of all edges
  - $T_i \rightarrow T_j$  for which one of three conditions holds:
    1.  $T_i$  executes write(Q) before  $T_j$  executes read(Q).
    2.  $T_i$  executes read(Q) before  $T_j$  executes write(Q).
    3.  $T_i$  executes write(Q) before  $T_j$  executes write(Q).



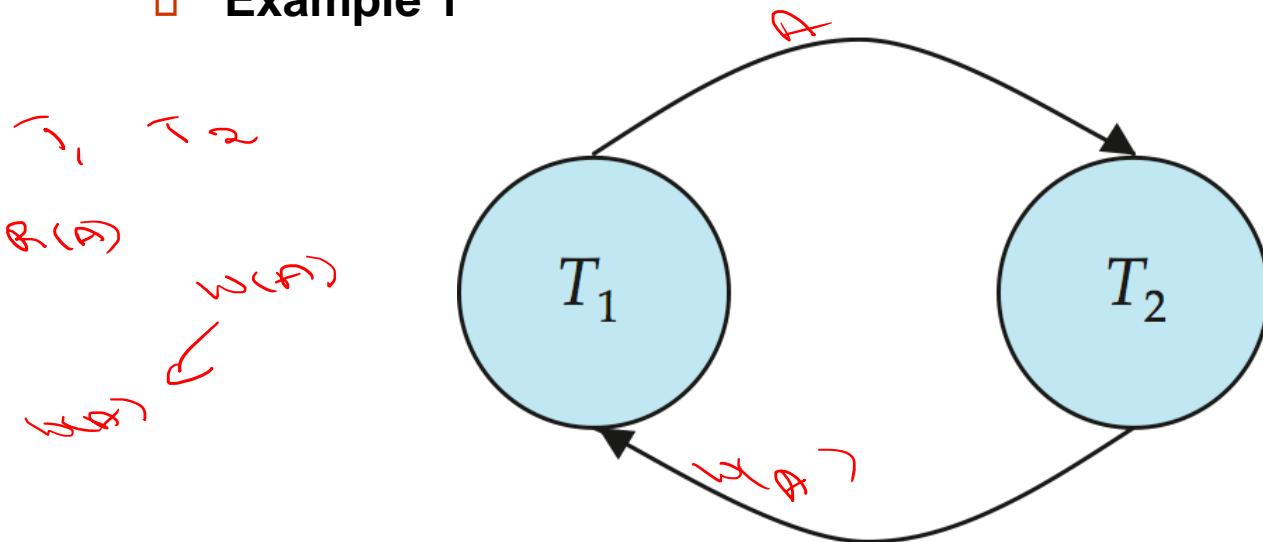
order -> transaction  $\rightarrow$  logical graph

order -> transaction



# Testing for Serializability

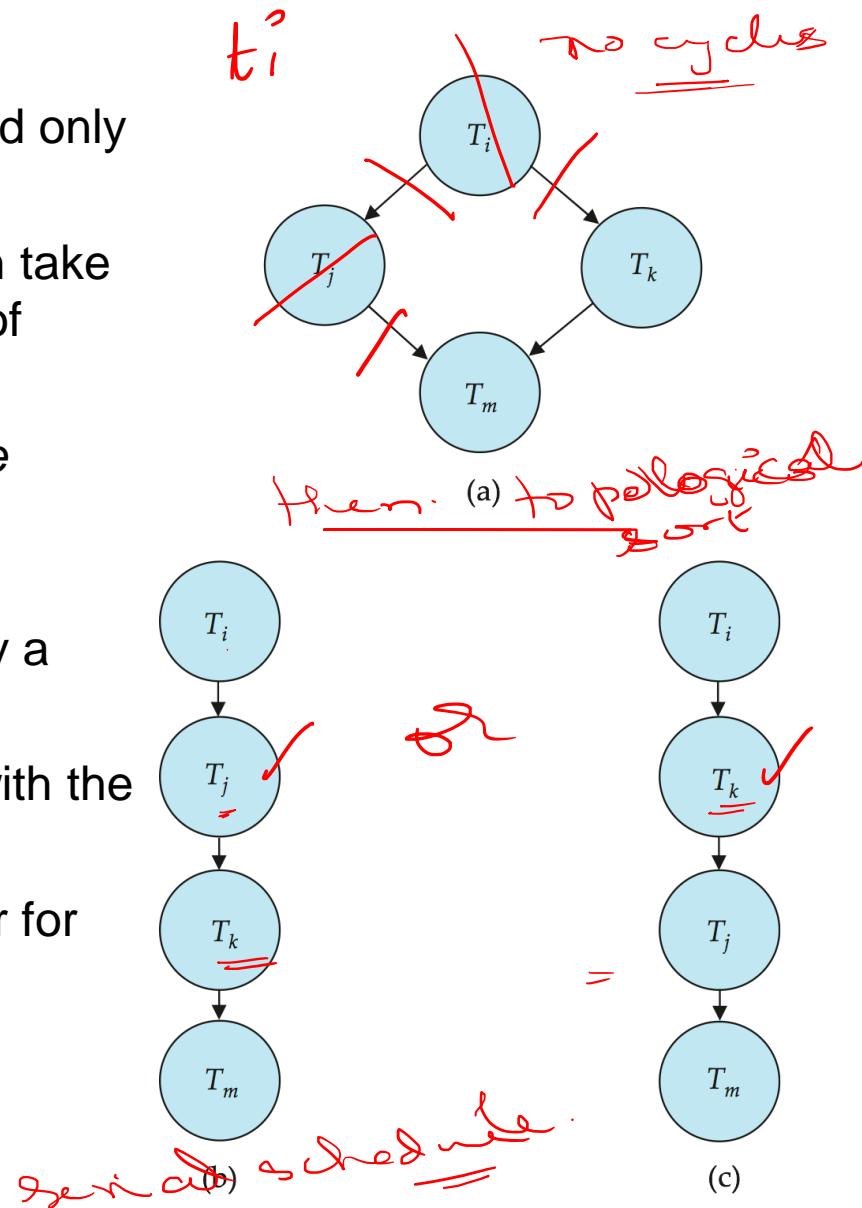
- Consider some schedule of a set of transactions  $T_1, T_2, \dots, T_n$
- **Precedence graph** — a directed graph where the vertices are the transactions (names).
- We draw an arc from  $T_i$  to  $T_j$  if the two transaction conflict, and  $T_i$  accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- **Example 1**





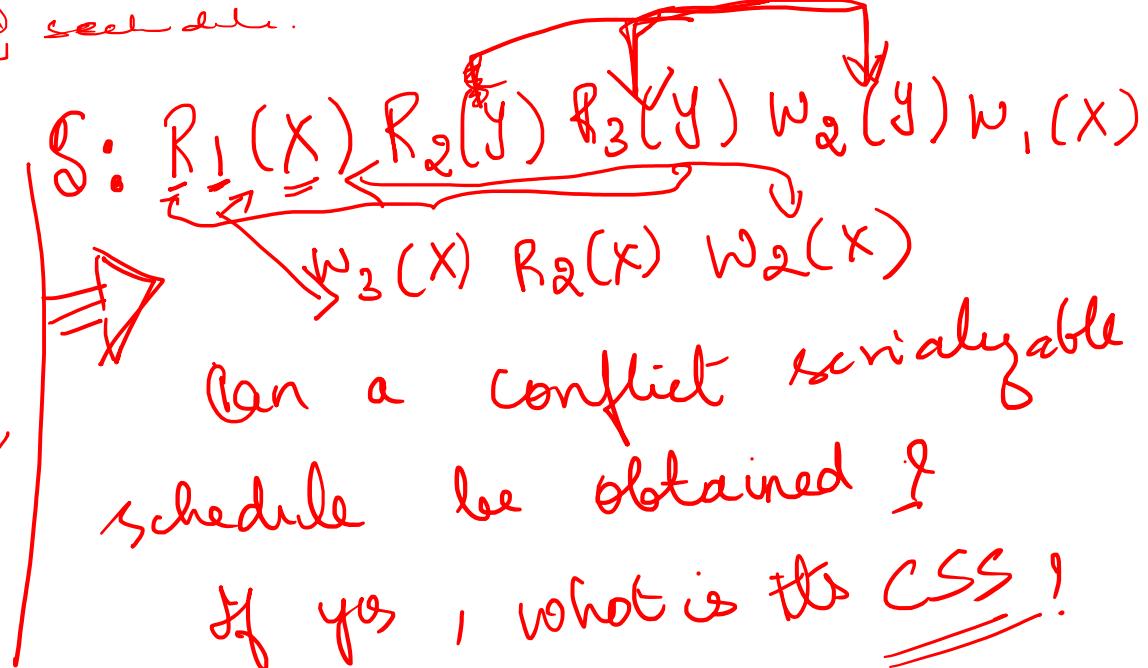
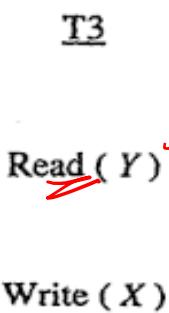
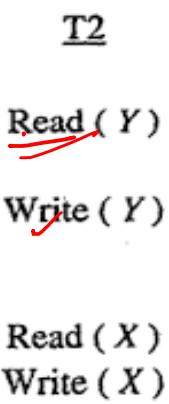
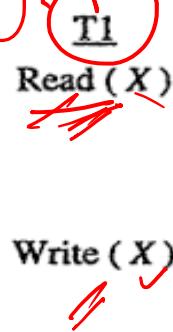
# Test for Conflict Serializability

- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order  $n^2$  time, where  $n$  is the number of vertices in the graph.
  - (Better algorithms take order  $n + e$  where  $e$  is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a topological sorting of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a serializability order for Schedule A would be
$$T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$$
    - ▶ Are there others?

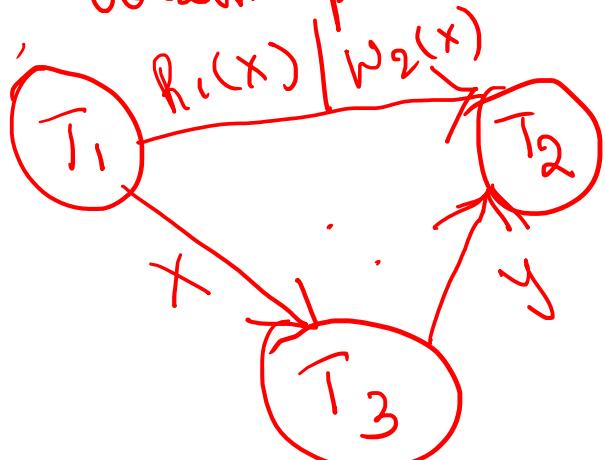




Concern  $\subseteq \Rightarrow$  serial schedule.

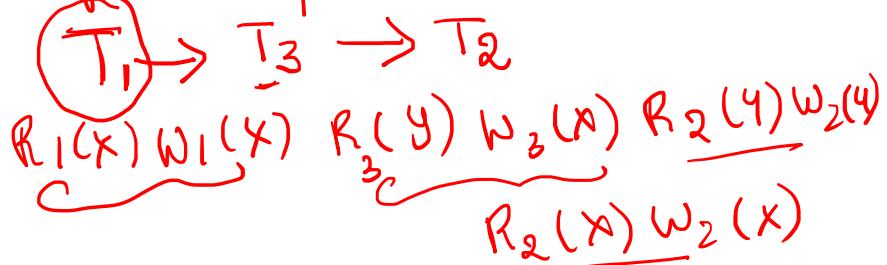


Step 1: obtain precedence graph



1) Yes, because Precedence Graph is acyclic

Step 2: topological sort



2 - 3



| $T_1$                         | $T_2$                         |
|-------------------------------|-------------------------------|
| read ( $A$ )<br>write ( $A$ ) | read ( $A$ )<br>write ( $A$ ) |
| read ( $B$ )<br>write ( $B$ ) | read ( $B$ )<br>write ( $B$ ) |

$\neq$



i) CSS : Yes  $\rightarrow$  no cycles

ii) solution :

$T_1 \sqsubset \sqsupseteq$

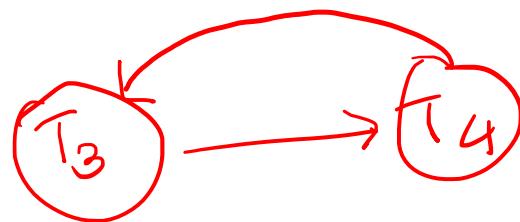
$\downarrow$

Full list of operations

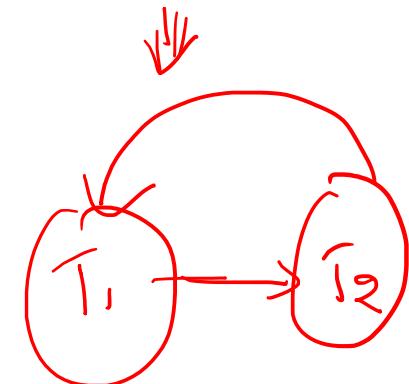
$\neq$



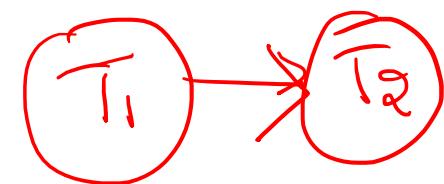
$S : R_3(Q) \rightsquigarrow W_4(Q) \rightsquigarrow W_3(Q)$



Cycles  $\therefore$  no CSS (an)



Ex:  $S : W_1(x) \rightsquigarrow W_2(x) \rightsquigarrow W_2(y) \rightsquigarrow W_1(y) \rightsquigarrow W_3(y)$



Precedence graph:  
Can a serial be obtained?

$W_1(x) \rightsquigarrow W_1(y) \rightsquigarrow W_2(x) \rightsquigarrow W_2(y) \rightsquigarrow W_3(y)$





Problem 1

| $T_1$    | $T_2$    | $T_3$    |
|----------|----------|----------|
| $w_1(A)$ |          |          |
|          | $R_2(A)$ |          |
|          |          | $w_3(A)$ |
|          |          |          |
|          | $w_3(A)$ |          |
|          |          |          |
|          | $w_1(A)$ |          |
| $S_1$    |          |          |

Problem 2

| $T_1$ | $T_2$    | $T_3$    |
|-------|----------|----------|
|       |          | $w_2(A)$ |
|       |          |          |
|       | $R_2(A)$ |          |
|       |          | $w_3(A)$ |
|       |          |          |
|       | $w_1(A)$ |          |
| $S_2$ |          |          |

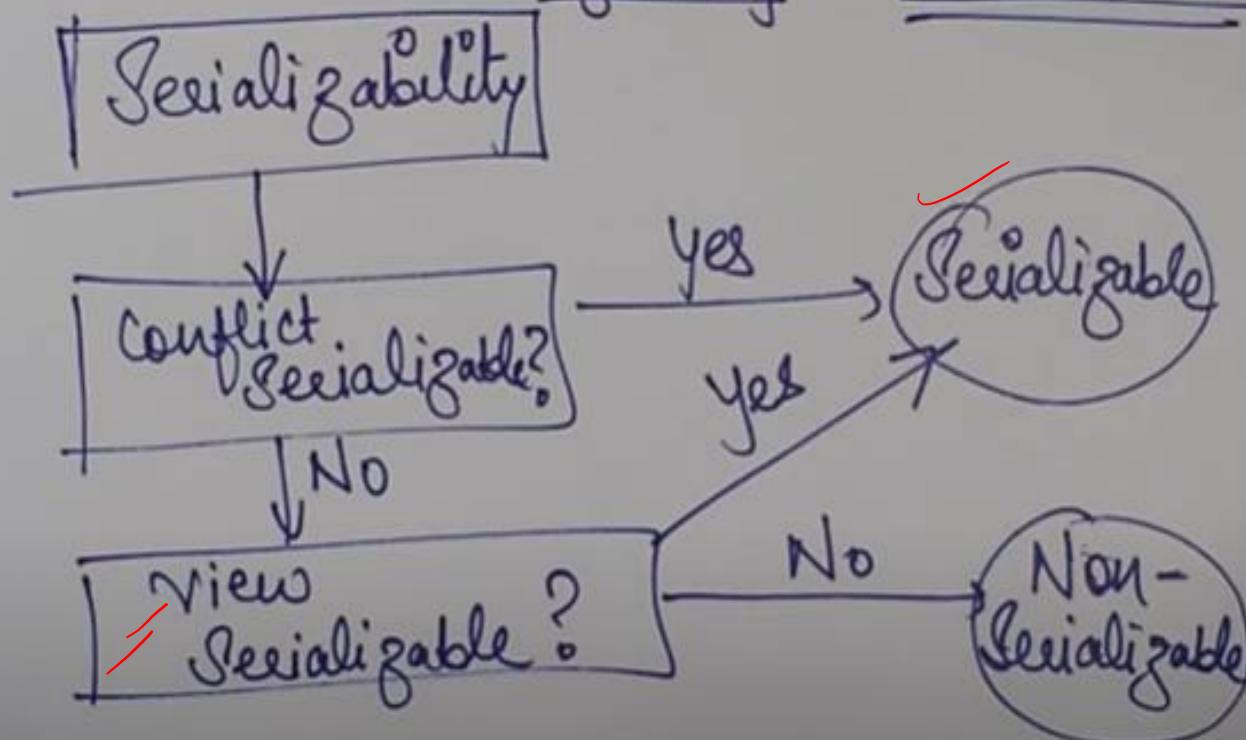
~~Without~~ is  $S_1$  conflict serializable  
vs  $S_2$  conflict serializable

$S_1$ : ~~list~~ :- serializable

$S_2$ : not serializable



## Serializability : A Broader View





# View Serializability

$S_1$  is concurrent schedule,  $S_2$  is your serial schedule

♦ Schedules  $S_1$  and  $S_2$  are view equivalent if:

- If  $T_i$  reads initial value of A in  $S_1$ , then  $T_i$  also reads initial value of A in  $S_2$
- If  $T_i$  reads value of A written by  $T_j$  in  $S_1$ , then  $T_i$  also reads value of A written by  $T_j$  in  $S_2$
- If  $T_i$  writes final update of A in  $S_1$ , then  $T_i$  also writes final value of A in  $S_2$

✓

|          |      |
|----------|------|
| T1: R(A) | W(A) |
| T2:      | W(A) |
| T3:      |      |

$S_1$

|                |      |
|----------------|------|
| T1: R(A), W(A) |      |
| T2:            | W(A) |
| T3:            | W(A) |

$S_2$





# View Serializability (Cont.)

- A schedule  $S$  is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

| $T_{27}$      | $T_{28}$      | $T_{29}$      |
|---------------|---------------|---------------|
| read ( $Q$ )  | write ( $Q$ ) |               |
| write ( $Q$ ) |               | write ( $Q$ ) |

- Every view serializable schedule that is not conflict serializable has **blind writes**.  
*without reading the item, the write takes place*

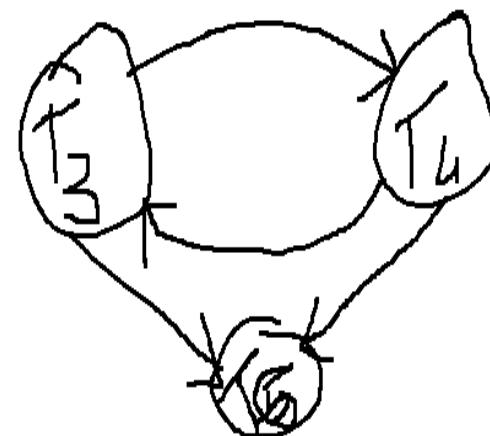


# ex1

| $T_3$       | $T_4$        | $T_6$ |
|-------------|--------------|-------|
| read( $Q$ ) |              |       |
|             | write( $Q$ ) |       |

Is it view Serializable?

1. Initial read
  2. Final update
  3. writes
- $a$
- $T_3 \rightarrow T_4 \rightarrow T_6$



Cycles  
not Conf.

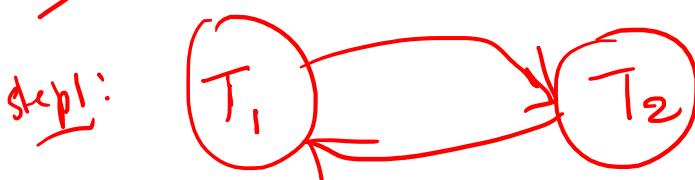


S:  $R_1(A)$   $w_2(A)$   $\uparrow$   $w_1(A)$   $\downarrow$   $w_3(A)$

Q:- Is this schedule view serializable:

If yes, give its view serializable / equivalent schedule

Ans :- Conflict serializable:



$T_{12}$  is not conflict serializable.

step2:-

$T_3$

} Initial Read  
Final update  
all writes

| A                 |   |
|-------------------|---|
| $T_1$             | ✓ |
| $T_3$             | ✓ |
| $T_2$ $T_1$ $T_3$ |   |

$A: T_1 \rightarrow T_2 \rightarrow T_3$

$R_1(A) W_1(A) W_2(A)$   
 $W_3(A)$



View serializability

Serial schedule



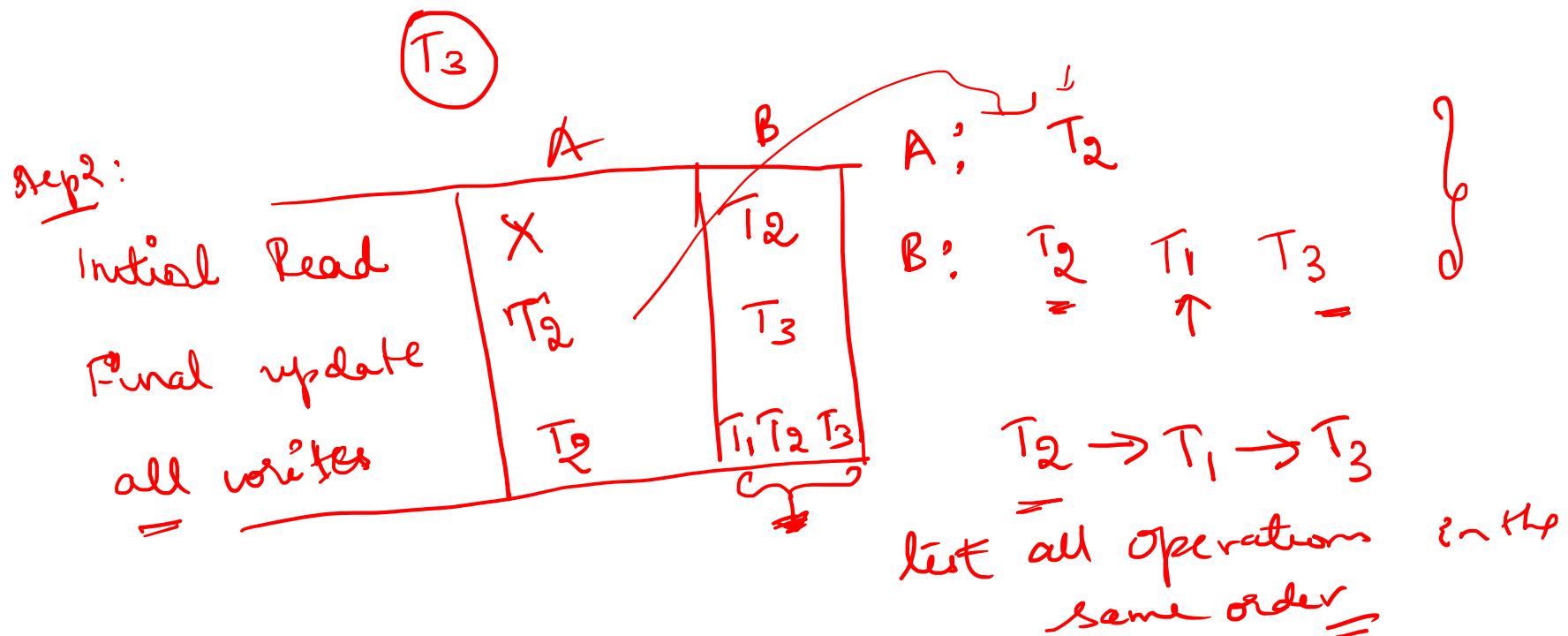


~~S~~:  $R_2(B)$   $W_2(A)$   $R_1(A)$   $R_3(A)$   $W_1(B)$   $W_2(B)$   $W_3(B)$

Sol: Precedence graph:



not conflict serializable

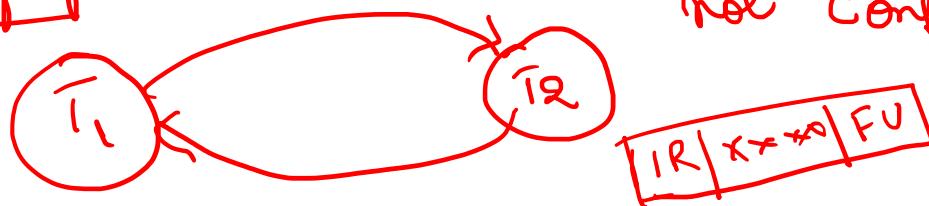




Ex3:  $S: R_1(A) R_2(B) Na(A)$        $R_2(B) W_1(A) R'_1(B) W_1(B)$

$\underbrace{R_1(A) R_2(B)}_{\text{conflict}}$        $\underbrace{R'_1(B) W_1(A)}_{\text{conflict}}$       <---

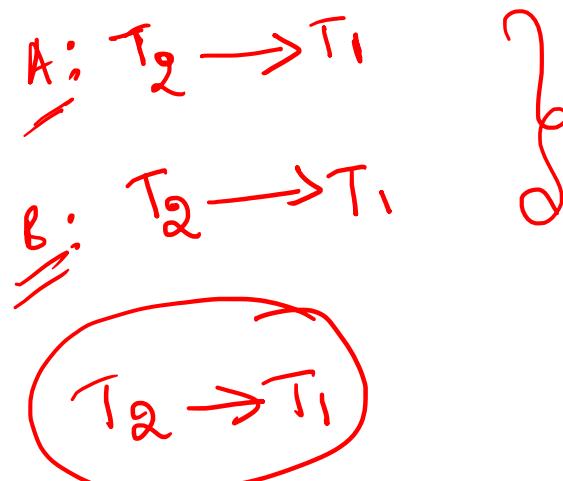
i) Precedence graph



not Conflict serializable

{ Initial Read      Final update      all writes }

|  | A          | B          |
|--|------------|------------|
|  | $T_1, T_2$ | $T_2, T_1$ |
|  | $T_1$      | $T_1$      |
|  | $T_2, T_1$ | $T_1$      |
|  |            | $T_1$      |



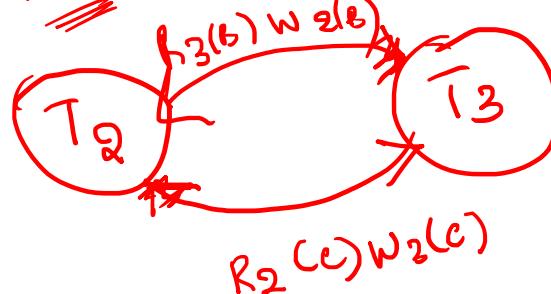
list all operations of  $T_2 \sqcap T_1$

---

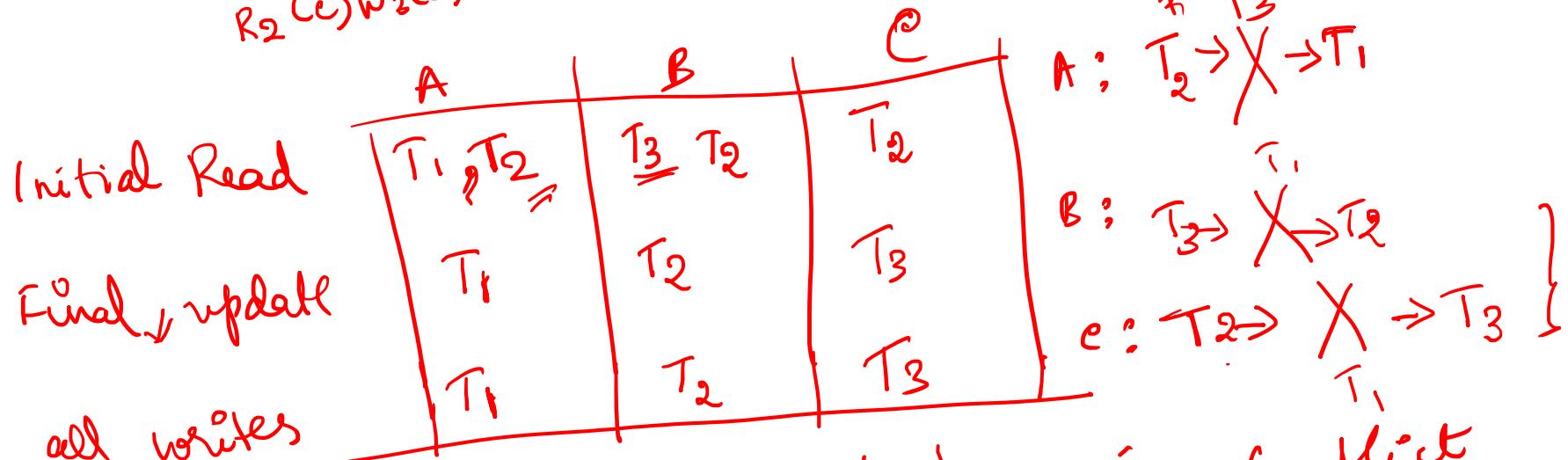


Ex:-

$\delta : R_1(A) R_2(A) R_3(B) W_1(A) \underline{R_2(C)} \quad \underline{R_2(B)} \quad W_2(B) \quad W_3(e)$



No CS-S possible



Ordering on 'B' and 'C' are in conflict

No solutions possible:  $\rightarrow$  No view  
serializable schedule is possible.





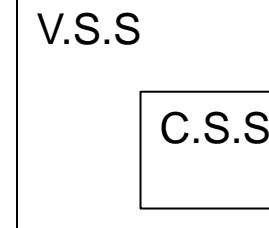


# ex3

Example:

| T1   | T2   |
|------|------|
| R(A) |      |
|      | R(A) |
|      | W(A) |
|      | R(B) |
| W(A) |      |
| R(B) |      |
| W(B) |      |

Set of all Schedules



Check for View serializable with  $\langle T1, T2 \rangle$  and  $\langle T2, T1 \rangle$



$T_1$      $T_2$   
 $R(A)$   
 $R(A)$   
 $W(A)$   
 $R(B)$   
 $W(A)$   
 $R(B)$   
 $W(B)$

1. Conflict Serializable

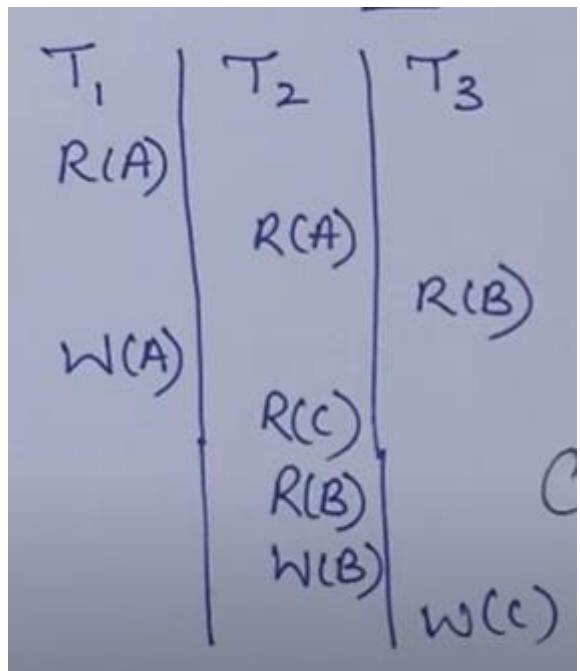


2. View Serializable

|                    | A                    | B                |                |
|--------------------|----------------------|------------------|----------------|
| Initial read       | $T_1 T_2$            | $T_2 T_1$        | A: $T_2 T_1$ / |
| Final update write | $T_1$                | $T_1$            | B: $T_2 T_1$ / |
|                    | $\overline{T_2 T_1}$ | $\overline{T_1}$ | $T_2 T_1$      |

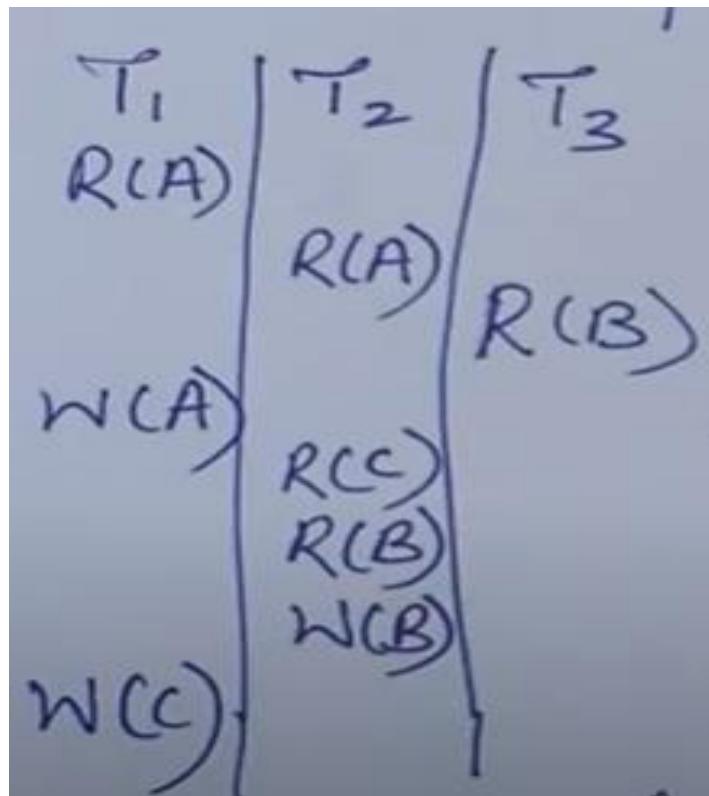


# Ex4: no serializable schedule possible





# Ex5: Serializable schedule possible





# For student workout

- Example: Check for View equivalence

T1

R(A)

W(A)

T2

R(A)

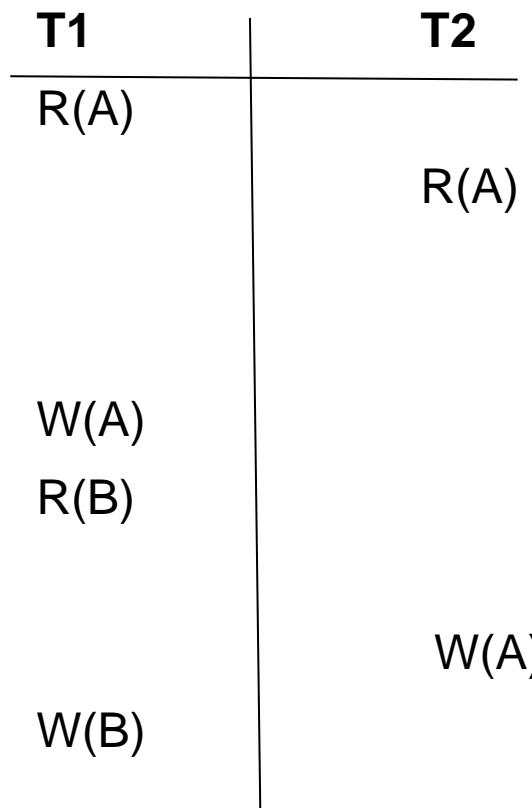
W(A)

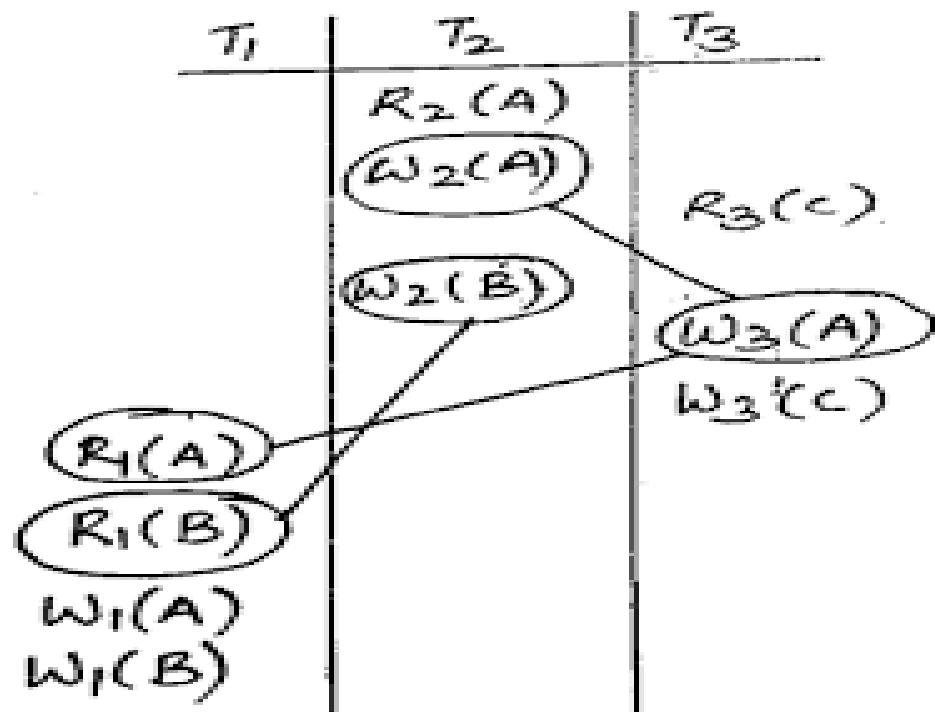
R(B)

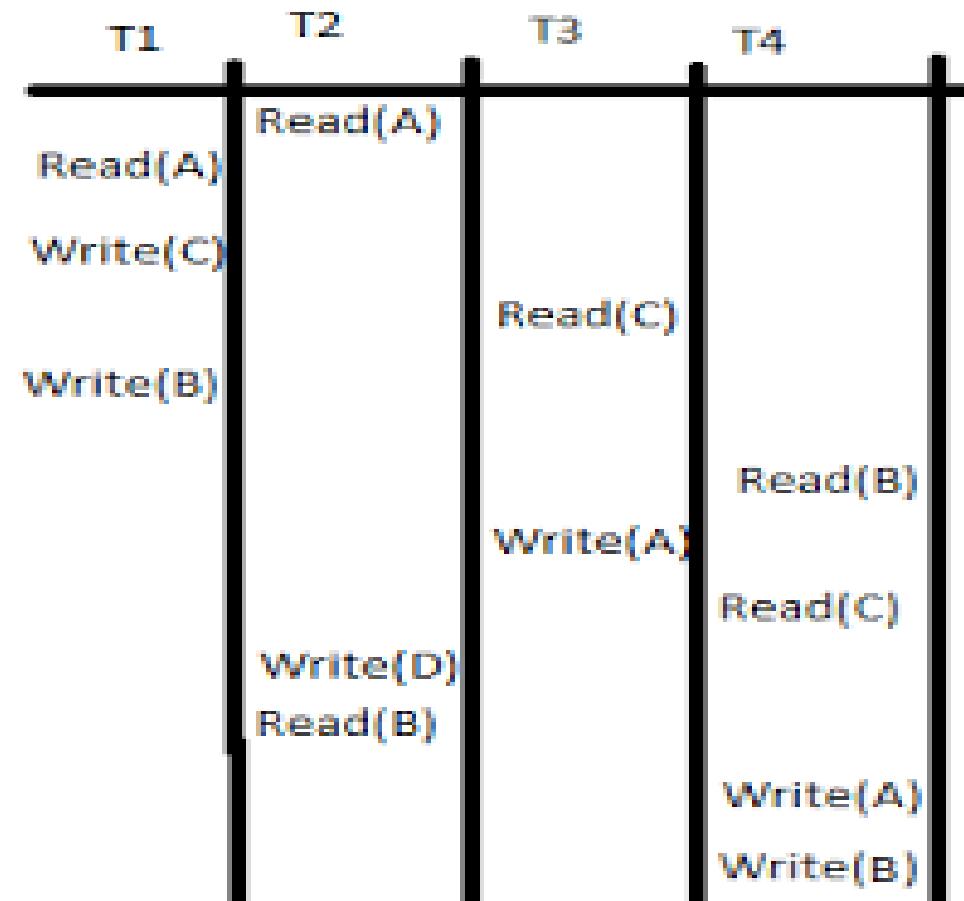
W(B)

R(B)

W(B)





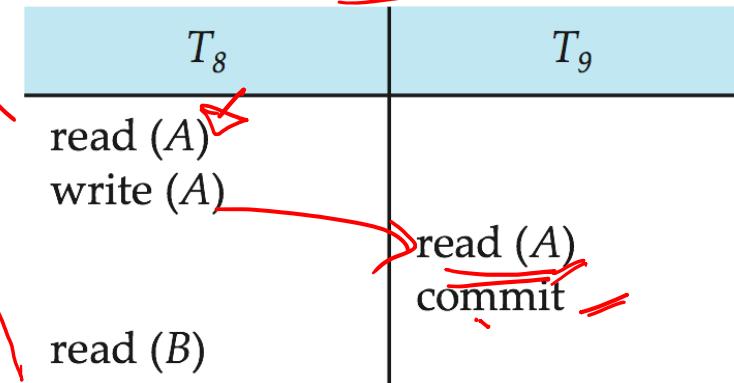
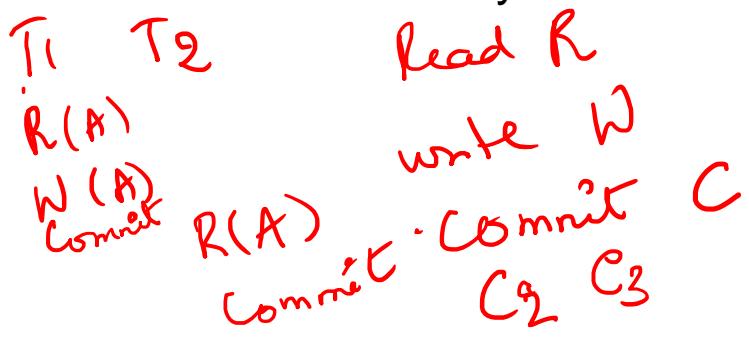




# Recoverable Schedules

Need to address the effect of transaction failures on concurrently running transactions.

- **Recoverable schedule** — if a transaction  $T_j$  reads a data item previously written by a transaction  $T_i$ , then the commit operation of  $T_i$  appears before the commit operation of  $T_j$ .
- The following schedule (Schedule 11) is not recoverable if  $T_9$  commits immediately after the read



- If  $T_8$  should abort,  $T_9$  would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.



# Cascading Rollbacks

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

*No commit*

| $T_{10}$                          | $T_{11}$                       | $T_{12}$                       |
|-----------------------------------|--------------------------------|--------------------------------|
| read (A)<br>read (B)<br>write (A) | read (A)<br>write (A)          | read (A)                       |
| abort                             | <i>updated</i><br><i>abort</i> | <i>updated</i><br><i>abort</i> |

- If  $T_{10}$  fails,  $T_{11}$  and  $T_{12}$  must also be rolled back.
- Can lead to the **undoing of a significant amount of work**



# Cascadeless Schedules

- **Cascade-less schedules** — cascading rollbacks cannot occur; for each pair of transactions  $T_i$  and  $T_j$  such that  $T_j$  reads a data item previously written by  $T_i$ , the commit operation of  $T_i$  appears before the read operation of  $T_j$ .
- Every cascade-less schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascade-less



# Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
- Testing a schedule for serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.



# Concurrency Control vs. Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless .
- Concurrency control protocols generally do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids nonserializable schedules.
  - We study such protocols in Chapter 16.
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for serializability help us understand why a concurrency control protocol is correct.



# Levels of Consistency in SQL-92

- **Serializable** — default
- **Repeatable read** — only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- **Read committed** — only committed records can be read, but successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.
  
- Lower degrees of consistency useful for gathering approximate information about the database
- Warning: some database systems do not ensure serializable schedules by default
  - E.g. Oracle and PostgreSQL by default support a level of consistency called snapshot isolation (not part of the SQL standard)



# Transaction Definition in SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off by a database directive
    - ▶ E.g. in JDBC, connection.setAutoCommit(false);



# End of Chapter 14

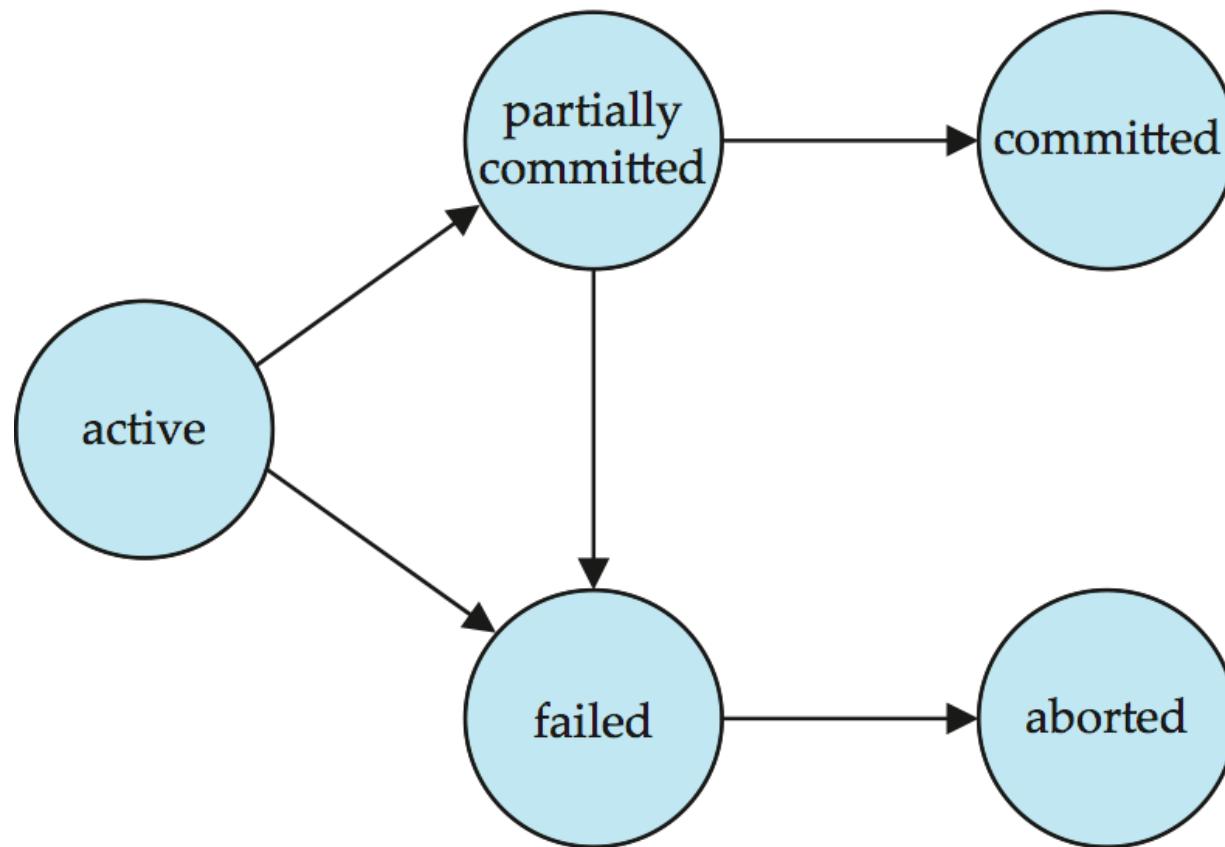
Database System Concepts, 6<sup>th</sup> Ed.

©Silberschatz, Korth and Sudarshan

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



# Figure 14.01





# Figure 14.02

| $T_1$                                                                                                      | $T_2$                                                                                                                               |
|------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| read ( $A$ )<br>$A := A - 50$<br>write ( $A$ )<br>read ( $B$ )<br>$B := B + 50$<br>write ( $B$ )<br>commit | read ( $A$ )<br>$temp := A * 0.1$<br>$A := A - temp$<br>write ( $A$ )<br>read ( $B$ )<br>$B := B + temp$<br>write ( $B$ )<br>commit |



# Figure 14.03

| $T_1$                                                                                                                                                                                                                                                                                                                                                                                                    | $T_2$ |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------|
| <p>read (<math>A</math>)<br/><math>temp := A * 0.1</math><br/><math>A := A - temp</math><br/>write (<math>A</math>)<br/>read (<math>B</math>)<br/><math>B := B + temp</math><br/>write (<math>B</math>)<br/>commit</p> <p>read (<math>A</math>)<br/><math>A := A - 50</math><br/>write (<math>A</math>)<br/>read (<math>B</math>)<br/><math>B := B + 50</math><br/>write (<math>B</math>)<br/>commit</p> |       |



# Figure 14.04

| $T_1$                                                                                                          | $T_2$                                                                                                                                   |
|----------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| read ( $A$ )<br>$A := A - 50$<br>write ( $A$ )<br><br>read ( $B$ )<br>$B := B + 50$<br>write ( $B$ )<br>commit | read ( $A$ )<br>$temp := A * 0.1$<br>$A := A - temp$<br>write ( $A$ )<br><br>read ( $B$ )<br>$B := B + temp$<br>write ( $B$ )<br>commit |



# Figure 14.05

| $T_1$                                                                     | $T_2$                                                                                 |
|---------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| read ( $A$ )<br>$A := A - 50$                                             | read ( $A$ )<br>$temp := A * 0.1$<br>$A := A - temp$<br>write ( $A$ )<br>read ( $B$ ) |
| write ( $A$ )<br>read ( $B$ )<br>$B := B + 50$<br>write ( $B$ )<br>commit | $B := B + temp$<br>write ( $B$ )<br>commit                                            |



# Figure 14.06

| $T_1$                         | $T_2$                         |
|-------------------------------|-------------------------------|
| read ( $A$ )<br>write ( $A$ ) | read ( $A$ )<br>write ( $A$ ) |
| read ( $B$ )<br>write ( $B$ ) | read ( $B$ )<br>write ( $B$ ) |



# Figure 14.07

| $T_1$         | $T_2$         |
|---------------|---------------|
| read ( $A$ )  |               |
| write ( $A$ ) |               |
| read ( $B$ )  | read ( $A$ )  |
| write ( $B$ ) | write ( $A$ ) |
|               | read ( $B$ )  |
|               | write ( $B$ ) |



# Figure 14.08

| $T_1$         | $T_2$         |
|---------------|---------------|
| read ( $A$ )  |               |
| write ( $A$ ) |               |
| read ( $B$ )  |               |
| write ( $B$ ) | read ( $A$ )  |
|               | write ( $A$ ) |
|               | read ( $B$ )  |
|               | write ( $B$ ) |

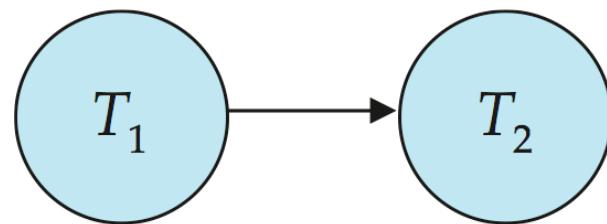


# Figure 14.09

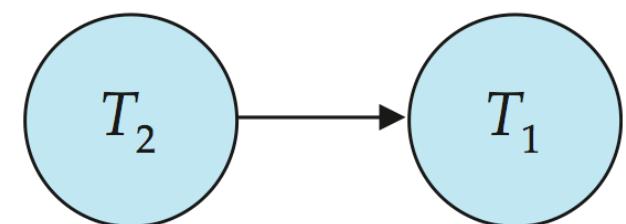
| $T_3$         | $T_4$         |
|---------------|---------------|
| read ( $Q$ )  |               |
| write ( $Q$ ) | write ( $Q$ ) |



# Figure 14.10



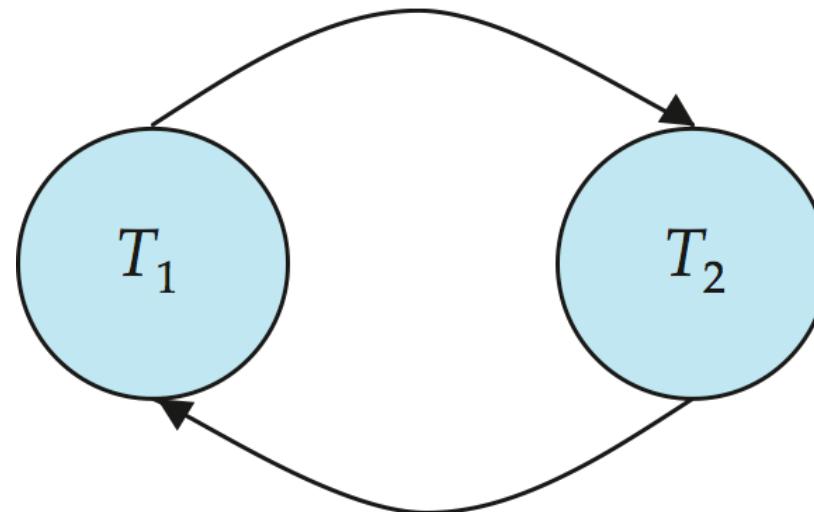
(a)



(b)

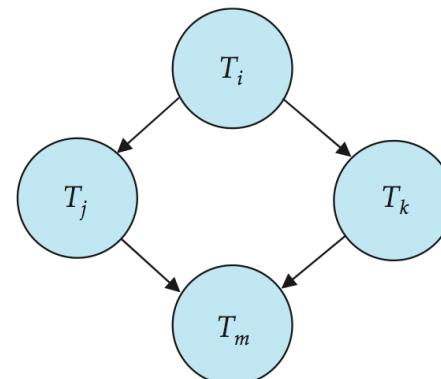


# Figure 14.11

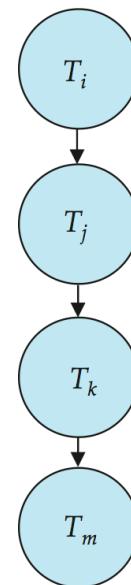




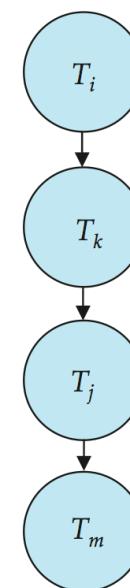
# Figure 14.12



(a)



(b)



(c)



# Figure 14.13

| $T_1$                                          | $T_5$                                          |
|------------------------------------------------|------------------------------------------------|
| read ( $A$ )<br>$A := A - 50$<br>write ( $A$ ) | read ( $B$ )<br>$B := B - 10$<br>write ( $B$ ) |
| read ( $B$ )<br>$B := B + 50$<br>write ( $B$ ) | read ( $A$ )<br>$A := A + 10$<br>write ( $A$ ) |



# Figure 14.14

| $T_8$                         | $T_9$                  |
|-------------------------------|------------------------|
| read ( $A$ )<br>write ( $A$ ) |                        |
| read ( $B$ )                  | read ( $A$ )<br>commit |

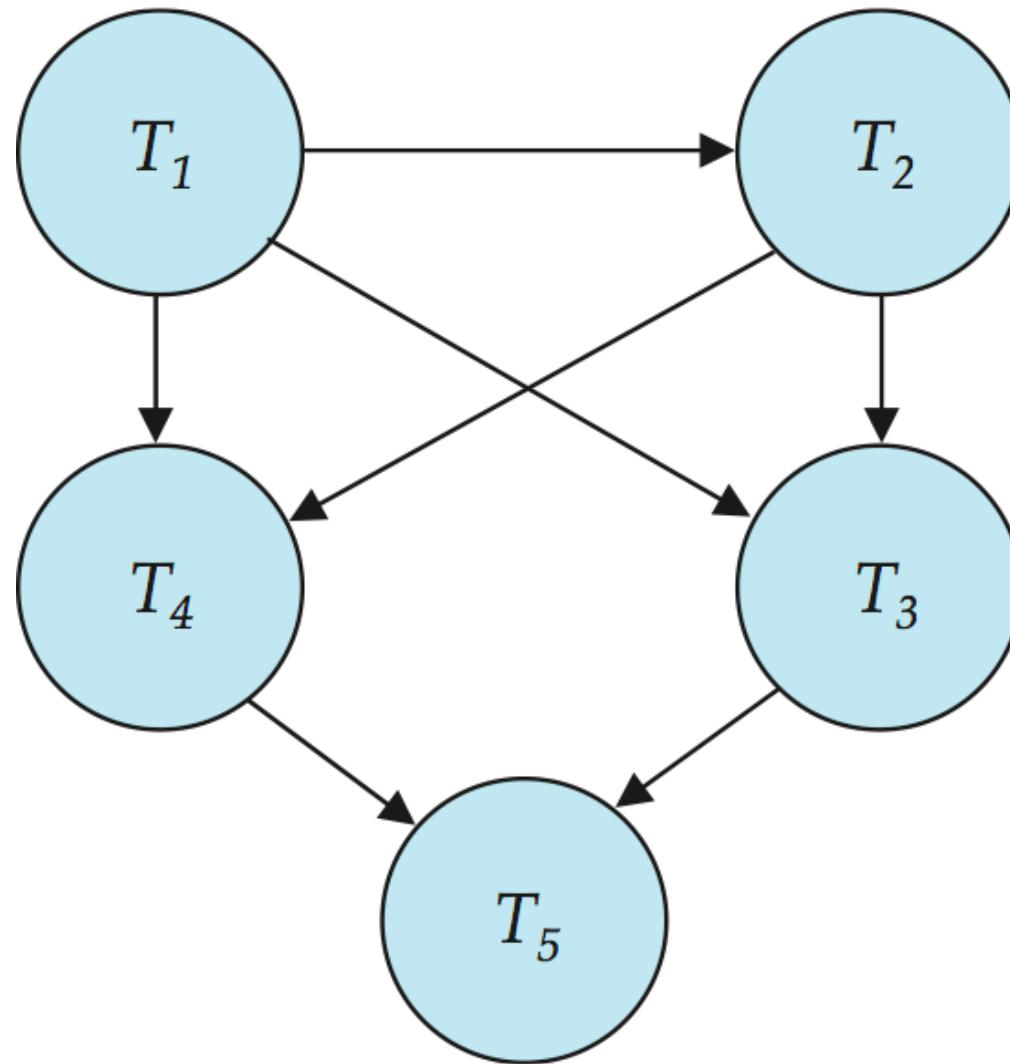


# Figure 14.15

| $T_{10}$                                                   | $T_{11}$                      | $T_{12}$     |
|------------------------------------------------------------|-------------------------------|--------------|
| read ( $A$ )<br>read ( $B$ )<br>write ( $A$ )<br><br>abort | read ( $A$ )<br>write ( $A$ ) | read ( $A$ ) |
|                                                            |                               |              |



# Figure 14.16





# End of Chapter 14

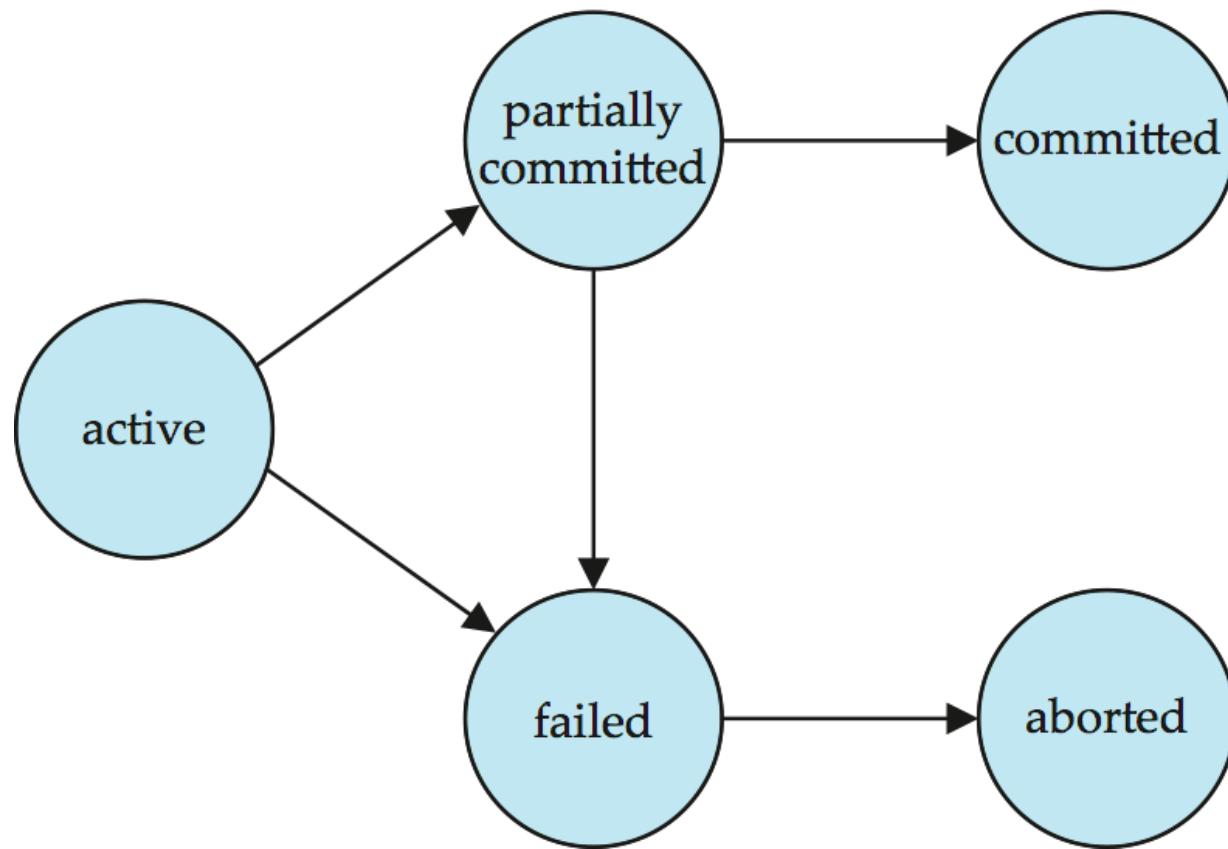
Database System Concepts, 6<sup>th</sup> Ed.

©Silberschatz, Korth and Sudarshan

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



# Figure 14.01





# Figure 14.02

| $T_1$                                                                                                      | $T_2$                                                                                                                               |
|------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| read ( $A$ )<br>$A := A - 50$<br>write ( $A$ )<br>read ( $B$ )<br>$B := B + 50$<br>write ( $B$ )<br>commit | read ( $A$ )<br>$temp := A * 0.1$<br>$A := A - temp$<br>write ( $A$ )<br>read ( $B$ )<br>$B := B + temp$<br>write ( $B$ )<br>commit |



# Figure 14.03

| $T_1$                                                                           | $T_2$                                                                                               |
|---------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------|
| <pre>read (A) A := A - 50 write (A) read (B) B := B + 50 write (B) commit</pre> | <pre>read (A) temp := A * 0.1 A := A - temp write (A) read (B) B := B + temp write (B) commit</pre> |



# Figure 14.04

| $T_1$                                                                                                          | $T_2$                                                                                                                                   |
|----------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| read ( $A$ )<br>$A := A - 50$<br>write ( $A$ )<br><br>read ( $B$ )<br>$B := B + 50$<br>write ( $B$ )<br>commit | read ( $A$ )<br>$temp := A * 0.1$<br>$A := A - temp$<br>write ( $A$ )<br><br>read ( $B$ )<br>$B := B + temp$<br>write ( $B$ )<br>commit |



# Figure 14.05

| $T_1$                                                                     | $T_2$                                                                                 |
|---------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| read ( $A$ )<br>$A := A - 50$                                             | read ( $A$ )<br>$temp := A * 0.1$<br>$A := A - temp$<br>write ( $A$ )<br>read ( $B$ ) |
| write ( $A$ )<br>read ( $B$ )<br>$B := B + 50$<br>write ( $B$ )<br>commit | $B := B + temp$<br>write ( $B$ )<br>commit                                            |



# Figure 14.06

| $T_1$                         | $T_2$                         |
|-------------------------------|-------------------------------|
| read ( $A$ )<br>write ( $A$ ) | read ( $A$ )<br>write ( $A$ ) |
| read ( $B$ )<br>write ( $B$ ) | read ( $B$ )<br>write ( $B$ ) |



# Figure 14.07

| $T_1$         | $T_2$         |
|---------------|---------------|
| read ( $A$ )  |               |
| write ( $A$ ) |               |
| read ( $B$ )  | read ( $A$ )  |
| write ( $B$ ) | write ( $A$ ) |
|               | read ( $B$ )  |
|               | write ( $B$ ) |



# Figure 14.08

| $T_1$         | $T_2$         |
|---------------|---------------|
| read ( $A$ )  |               |
| write ( $A$ ) |               |
| read ( $B$ )  |               |
| write ( $B$ ) | read ( $A$ )  |
|               | write ( $A$ ) |
|               | read ( $B$ )  |
|               | write ( $B$ ) |

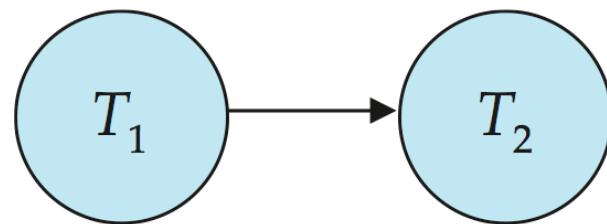


# Figure 14.09

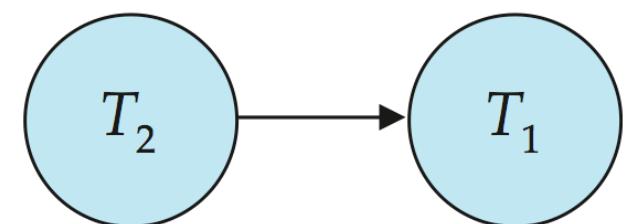
| $T_3$         | $T_4$         |
|---------------|---------------|
| read ( $Q$ )  |               |
| write ( $Q$ ) | write ( $Q$ ) |



# Figure 14.10



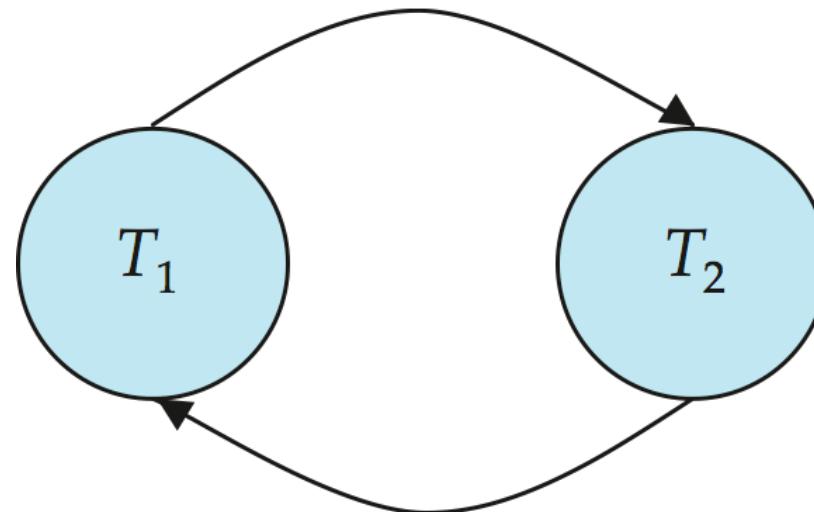
(a)



(b)

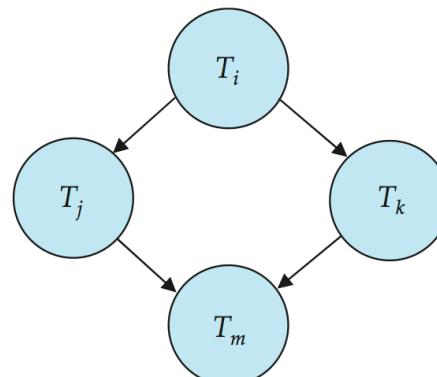


# Figure 14.11

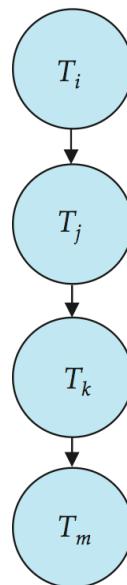




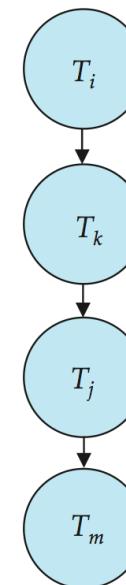
# Figure 14.12



(a)



(b)



(c)



# Figure 14.13

| $T_1$                                          | $T_5$                                          |
|------------------------------------------------|------------------------------------------------|
| read ( $A$ )<br>$A := A - 50$<br>write ( $A$ ) | read ( $B$ )<br>$B := B - 10$<br>write ( $B$ ) |
| read ( $B$ )<br>$B := B + 50$<br>write ( $B$ ) | read ( $A$ )<br>$A := A + 10$<br>write ( $A$ ) |



# Figure 14.14

| $T_8$                         | $T_9$                  |
|-------------------------------|------------------------|
| read ( $A$ )<br>write ( $A$ ) |                        |
| read ( $B$ )                  | read ( $A$ )<br>commit |

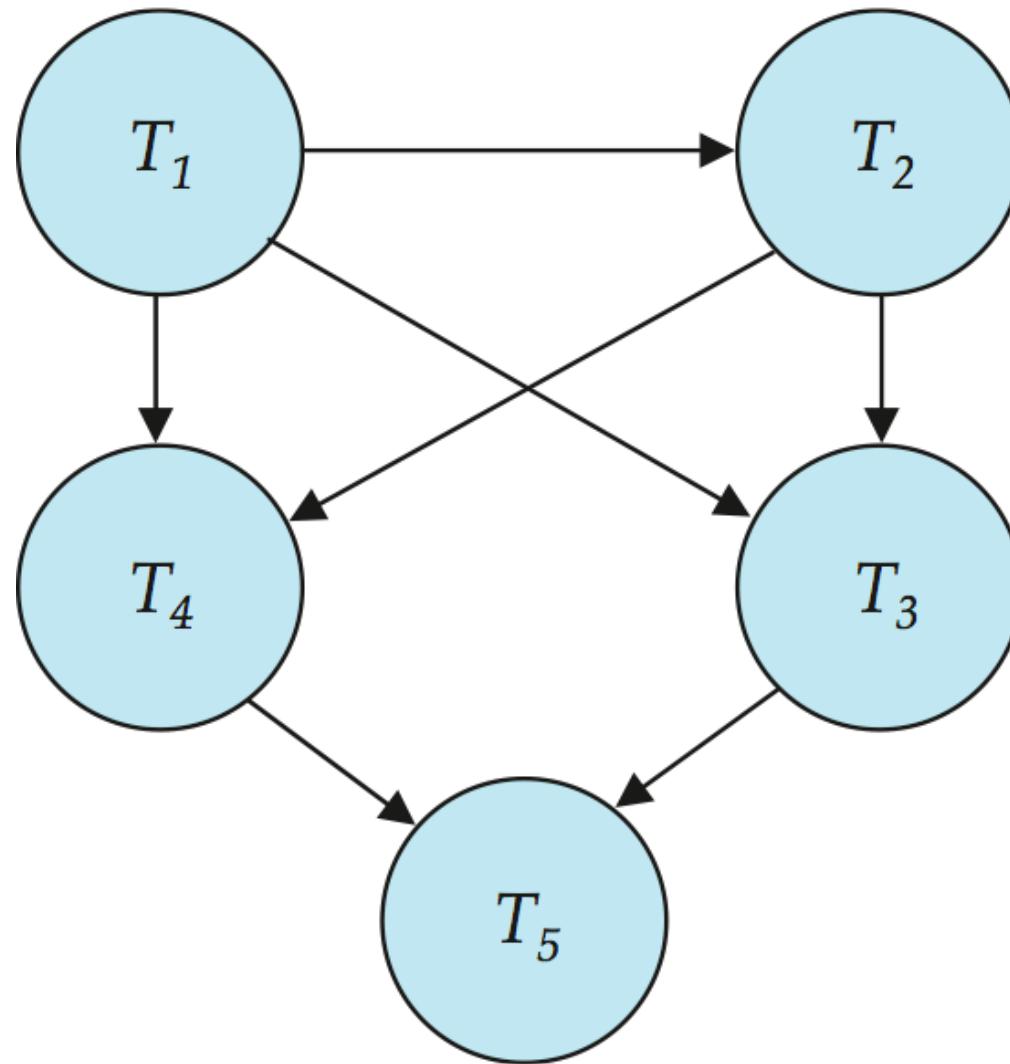


# Figure 14.15

| $T_{10}$                                                   | $T_{11}$                      | $T_{12}$     |
|------------------------------------------------------------|-------------------------------|--------------|
| read ( $A$ )<br>read ( $B$ )<br>write ( $A$ )<br><br>abort | read ( $A$ )<br>write ( $A$ ) | read ( $A$ ) |
|                                                            |                               |              |



# Figure 14.16



# Normal forms and MVD

Fd  
Fd<sup>+</sup>  
A<sup>+</sup>, Keys, Superkeys  
Canonical form  
Equivalence  
L $\leq$ , DPD

PPD

- **Normal Forms**
- Decomposition to 3NF
- Decomposition to BCNF

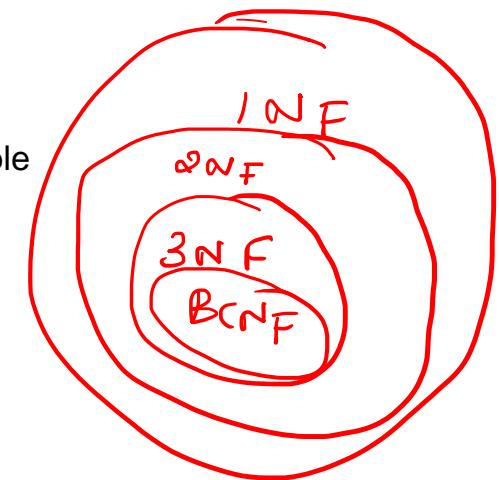
## NORMAL FORMS



# Normalization or Schema Refinement

PPD

- Normalization or Schema Refinement is a technique of organizing the data in the database
- A systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics
  - Insertion Anomaly
  - Update Anomaly
  - Deletion Anomaly
- Most common technique for the Schema Refinement is decomposition.
  - Goal of Normalization: Eliminate Redundancy
- Redundancy refers to repetition of same data or duplicate copies of same data stored in different locations
- Normalization is used for mainly two purpose:
  - Eliminating redundant (useless) data
  - Ensuring data dependencies make sense, that is, data is logically stored



# Anomalies *Issue*

- 1. Update Anomaly:** Employee 519 is shown as having different addresses on different records

**Employees' Skills**

| Employee ID | Employee Address   | Skill           |
|-------------|--------------------|-----------------|
| 426         | 57 Sycamore Grove  | Typing          |
| 426         | 57 Sycamore Grove  | Shorthand       |
| 519         | 34 Chestnut Street | Public Speaking |
| 519         | 36 Walnut Avenue   | Carpentry       |

**Resolution: Decompose the Schema**

1. **Update:** (ID, Address), (ID, Skill)
2. **Insert:** (ID, Name, Hire Date), (ID, Code)
3. **Delete:** (ID, Name, Hire Date), (ID, Code)

- 2. Insertion Anomaly:** Until the new faculty member, Dr. Newsome, is assigned to teach at least one course, his details cannot be recorded

**Faculty and Their Courses**

| Faculty ID | Faculty Name   | Faculty Hire Date | Course ID Code |
|------------|----------------|-------------------|----------------|
| 389        | Dr. Giddens    | 10-Feb-1985       | ENG-206        |
| 407        | Dr. Saperstein | 19-Apr-1999       | CMP-101        |
| 407        | Dr. Saperstein | 19-Apr-1999       | CMP-201        |
| 424        | Dr. Newsome    | 29-Mar-2007       | ?              |

- 3. Deletion Anomaly:** All information about Dr. Giddens is lost if he temporarily ceases to be assigned to any courses.

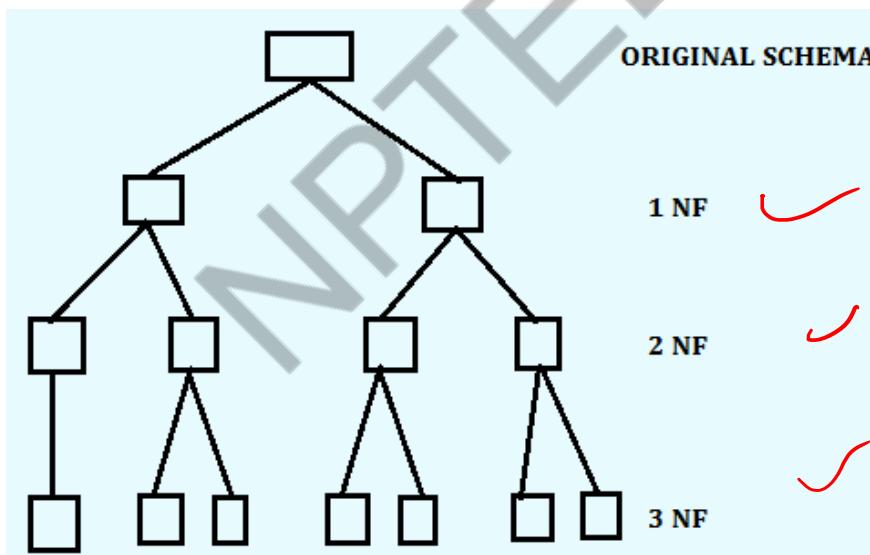
**Faculty and Their Courses**

| Faculty ID | Faculty Name   | Faculty Hire Date | Course ID Code |
|------------|----------------|-------------------|----------------|
| 389        | Dr. Giddens    | 10-Feb-1985       | ENG-206        |
| 407        | Dr. Saperstein | 19-Apr-1999       | CMP-101        |
| 407        | Dr. Saperstein | 19-Apr-1999       | CMP-201        |

# Desirable Properties of Decomposition

PPD

- Lossless Join Decomposition Property
  - It should be possible to reconstruct the original table
- Dependency Preserving Property
  - No functional dependency (or other constraints should get violated)



Boyce-Codd N.F  
BCNF  
3NF

# Normalization and Normal Forms

- A normal form specifies a set of conditions that the relational schema must satisfy in terms of its constraints – they offer varied levels of guarantee for the design
- Normalization rules are divided into various normal forms. Most common normal forms are:
  - First Normal Form (1 NF)
  - Second Normal Form (2 NF)
  - Third Normal Form (3 NF)
- Informally, a relational database relation is often described as "normalized" if it meets third normal form. Most 3NF relations are free of insertion, update, and deletion anomalies

# Normalization and Normal Forms

PPD

## ■ Additional Normal Forms

- Elementary Key Normal Form (EKNF)
- Boyce-codd Normal Form (BCNF)
- Multivalued Dependencies And Fourth Normal Form (4 NF)
- Essential Tuple Normal Form (ETNF)
- Join Dependencies And Fifth Normal Form (5 NF)
- Sixth Normal Form (6NF)
- Domain/Key Normal Form (DKNF)

not in syllabus

# First Normal Form (1 NF)

- A relation is in first Normal Form if and only if all underlying domains contain atomic values only
- In other words, a relation doesn't have multivalued attributes (MVA)
- Example:

• **STUDENT(Sid, Sname, Cname)**

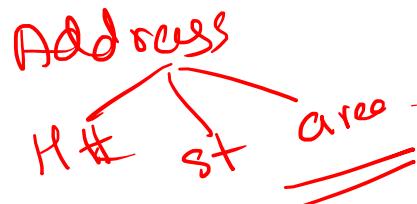
*no breaking the attributes*

*PK = C++ CPlus Plus*

| Students |       |         |
|----------|-------|---------|
| SID      | Sname | Cname   |
| S1       | A     | C,C++   |
| S2       | B     | C++, DB |
| S3       | A     | DB      |

| Students |       |       |
|----------|-------|-------|
| SID      | Sname | Cname |
| S1       | A     | C     |
| S1       | A     | C++   |
| S2       | B     | C++   |
| S2       | B     | DB    |
| S3       | A     | DB    |



## Example:

- Supplier(SID, Status, City, PID, Qty)**

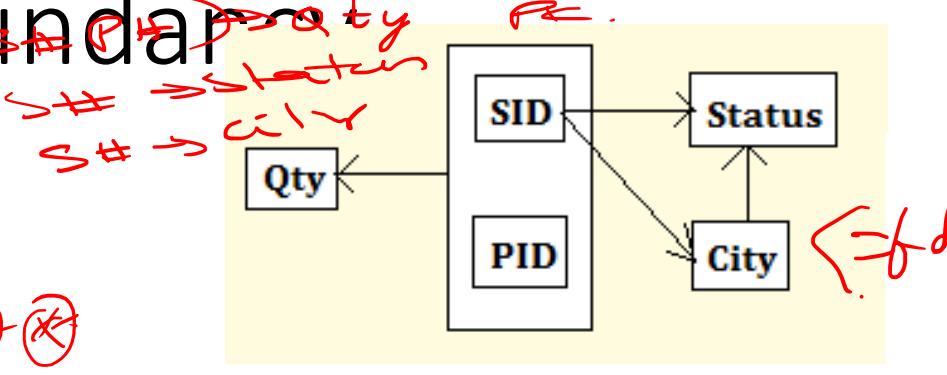
| Supplier: |        |        |     |     |
|-----------|--------|--------|-----|-----|
| SID       | Status | City   | PID | Qty |
| S1        | 30     | Delhi  | P1  | 100 |
| S1        | 30     | Delhi  | P2  | 125 |
| S1        | 30     | Delhi  | P3  | 200 |
| S1        | 30     | Delhi  | P4  | 130 |
| S2        | 10     | Karnal | P1  | 115 |
| S2        | 10     | Karnal | P2  | 250 |
| S3        | 40     | Rohtak | P1  | 245 |
| S4        | 30     | Delhi  | P4  | 300 |
| S4        | 30     | Delhi  | P5  | 315 |

Key : **(SID, PID)**

No MUA  
No composite  
 $\therefore$  1 NF  $\uparrow$  there are anomalies

# First Normal Form (1 NF):

PPD



### Drawbacks:

- Deletion Anomaly** – If we delete the tuple  $\langle S3, 40, \text{Rohtak}, P1, 245 \rangle$ , then we lose the information about S3 that S3 lives in Rohtak.
- Insertion Anomaly** – We cannot insert a Supplier S5 located in Karnal, until S5 supplies at least one part.
- Updation Anomaly** – If Supplier S1 moves from Delhi to Kanpur, then it is difficult to update all the tuples containing (S1, Delhi) as SID and City respectively.

Normal Forms are the methods of reducing redundancy. However, Sometimes 1 NF increases redundancy. It does not make any efforts in order to decrease redundancy.

# First Normal Form (1 NF):

PPD

## Possible Redundancy

### When LHS is not a Superkey :

- Let  $X \rightarrow Y$  is a non trivial FD over R with X is not a superkey of R, then redundancy exist between X and Y attribute set.
- Hence in order to identify the redundancy, we need not to look at the actual data, it can be identified by given functional dependency.
- Example :  $X \rightarrow Y$  and X is not a Candidate Key  
 $\Rightarrow X$  can duplicate  
 $\Rightarrow$  corresponding Y value would duplicate also.

| X | Y |
|---|---|
| 1 | 3 |
| 1 | 3 |
| 2 | 3 |
| 2 | 3 |
| 4 | 6 |

Fig 1

### When LHS is a Superkey :

- If  $X \rightarrow Y$  is a non trivial FD over R with X is a superkey of R, then redundancy does not exist between X and Y attribute set.
- Example :  $X \rightarrow Y$  and X is a Candidate Key  
 $\Rightarrow X$  cannot duplicate  
 $\Rightarrow$  corresponding Y value may or may not duplicate.



| X | Y |
|---|---|
| 1 | 4 |
| 2 | 6 |
| 3 | 4 |

Source: <http://www.edugrabs.com/normal-forms/#fnf>

# Second Normal Form (2 NF)

- Relation  $R$  is in Second Normal Form (2NF) only iff :
  - $R$  should be in 1NF and
  - $R$  should not contain any *Partial Dependency*

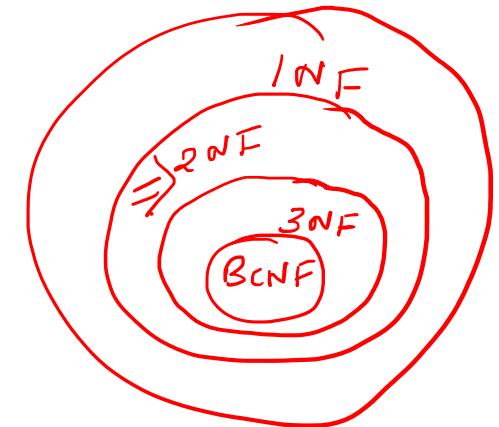
## Partial Dependency:

Let  $R$  be a relational Schema and  $X, Y, A$  be the attribute sets over  $R$  where  
 $\underline{X}$ : Any Candidate Key,  $\underline{Y}$ : Proper Subset of Candidate Key, and  $\underline{A}$ : Non Key Attribute

If  $Y \rightarrow A$  exists in  $R$ , then  $R$  is not in 2 NF.

$(Y \rightarrow A)$  is a Partial dependency only if

- $\underline{Y}$ : Proper subset of Candidate Key
- $\underline{A}$ : Non Prime Attribute



Source: <http://www.edugrabs.com/2nf-second-normal-form/>

# Second Normal Form (2 NF)

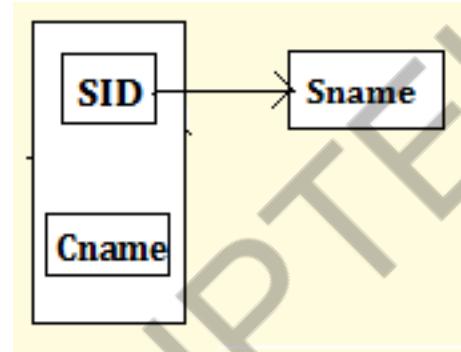
PPD

## Example:

- **STUDENT(Sid, Sname, Cname)** (already in 1NF)

| Students: |       |       |
|-----------|-------|-------|
| SID       | Sname | Cname |
| S1        | A     | C     |
| S1        | A     | C++   |
| S2        | B     | C++   |
| S2        | B     | DB    |
| S3        | A     | DB    |

(SID, Cname): Primary Key



### Functional Dependencies:

$$\{ \text{SID}, \text{Cname} \} \rightarrow \text{Sname}$$
$$\text{SID} \rightarrow \text{Sname}$$

### Partial Dependencies:

SID → Sname (as SID is a Proper Subset of Candidate Key {SID,Cname})

- **Redundancy?**
  - Sname
- **Anomaly?**
  - Yes

## Post Normalization

| R1: |       |
|-----|-------|
| SID | Sname |
| S1  | A     |
| S2  | B     |
| S3  | A     |

{SID}: Primary Key

| R2: |       |
|-----|-------|
| SID | Cname |
| S1  | C     |
| S1  | C++   |
| S2  | C++   |
| S2  | DB    |
| S3  | DB    |

{SID,Cname}: Primary Key

$\text{SID} \rightarrow \text{Sname}$

- The above two relations R1 and R2 are
1. Lossless Join
  2. 2NF
  3. Dependency Preserving

$1NF \rightarrow MVA$   
 $2NF \rightarrow P.D.$

| Supplier: |        |        |     |     |
|-----------|--------|--------|-----|-----|
| SID       | Status | City   | PID | Qty |
| S1        | 30     | Delhi  | P1  | 100 |
| S1        | 30     | Delhi  | P2  | 125 |
| S1        | 30     | Delhi  | P3  | 200 |
| S1        | 30     | Delhi  | P4  | 130 |
| S2        | 10     | Karnal | P1  | 115 |
| S2        | 10     | Karnal | P2  | 250 |
| S3        | 40     | Rohtak | P1  | 245 |
| S4        | 30     | Delhi  | P4  | 300 |
| S4        | 30     | Delhi  | P5  | 315 |

Key : (SID, PID)

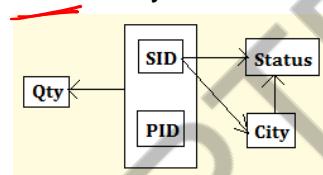
Source: <http://www.edugrabs.com/2nf-second-normal-form/>

# Second Normal Form (2 NF): Possible Redundancy

Example:

- Supplier(SID, Status, City, PID, Qty)

Partial Dependencies:  
 $SID \rightarrow Status$   
 $SID \rightarrow City$

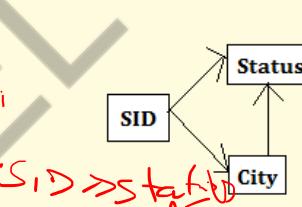


Not in 2NF

## Post Normalization

| Sup_City :          |
|---------------------|
| SID   Status   City |

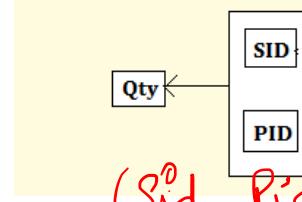
### FDD of Sup\_City :



## Sup\_Qty :

| Sup_Qty :       |
|-----------------|
| SID   PID   Qty |

### FDD of Sup\_qty :



$(SID, PID) \rightarrow Qty$

### Drawbacks:

- Deletion Anomaly** – If we delete a tuple in Sup\_City, then we not only lose the information about a supplier, but also lose the status value of a particular city.
- Insertion Anomaly** – We cannot insert a City and its status until a supplier supplies at least one part.
- Updation Anomaly** – If the status value for a city is changed, then we will face the problem of searching every tuple for that city.

$City \rightarrow Status$   
 $NPA \rightarrow NPA$

2NF

Candidate Key,  $? = \underline{\text{SID}}$

PPD

1  
6  
.1  
7  
D  
a  
@  
S  
a  
i  
b  
l  
a  
b  
s  
ee  
rs  
sy  
cs  
ht  
ae  
tm  
zc  
,o  
Kn  
oc  
re  
tp  
ht  
as  
n  
d  
d6  
S  
u  
d  
di  
t  
ai  
r  
o  
s  
h  
a

Reason for redundancy

PPD

# Second Normal Form (2 NF):

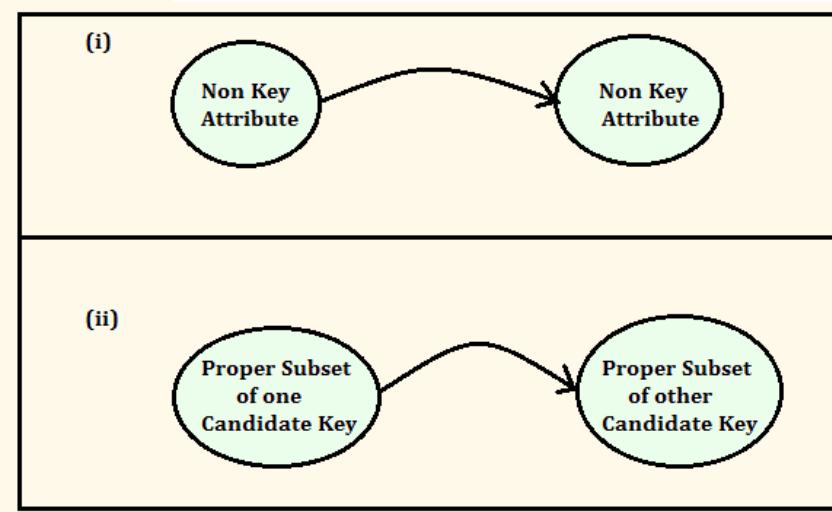
## Possible Redundancy

- In the **Sup\_City** relation :

- **City → Status**
    - Non Key Attribute → Non Key Attribute

- In the **STUDENT** relation:

- **SID → Cname**
    - Proper Subset of 1 CK → Proper Subset of other CK



Source: <http://www.edugrabs.com/2nf-second-normal-form/>

# Third Normal Form (3 NF)

LHS  $\rightarrow$  RHS  
<sub>PPD</sub>

AB  $\rightarrow$  B

$X \rightarrow A$   
CAB  $\rightarrow$  AD

$A - X$

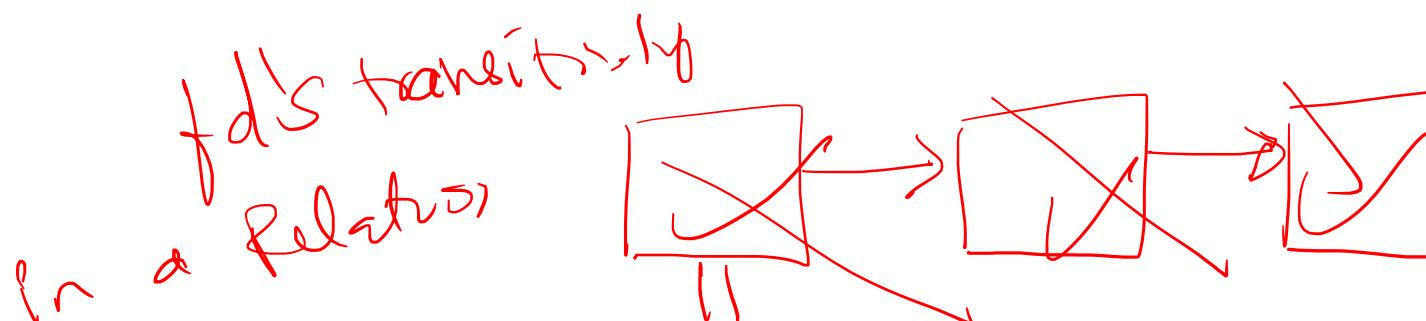
Let  $R$  be the relational schema.

- [E. F. Codd, 1971]  $R$  is in 3NF only if:
  - $R$  should be in 2NF
  - $R$  should not contain transitive dependencies (OR, Every non-prime attribute of  $R$  is non-transitively dependent on every key of  $R$ )
- [Carlo Zaniolo, 1982] Alternately,  $R$  is in 3NF iff for each of its functional dependencies  $X \rightarrow A$ , at least one of the following conditions holds:
  - $X$  contains  $A$  (that is,  $A$  is a subset of  $X$ , meaning  $X \rightarrow A$  is trivial functional dependency), or
  - $X$  is a superkey, or
  - Every element of  $A - X$ , the set difference between  $A$  and  $X$ , is a prime attribute (i.e., each attribute in  $A - X$  is contained in some candidate key)
- [Simple Statement] A relational schema  $R$  is in 3NF if for every FD  $X \rightarrow A$  associated with  $R$  either
  - $A \subseteq X$  (i.e., the FD is trivial) or
  - $X$  is a superkey of  $R$  or
  - $A$  is part of some key (not just superkey!)

Source: <http://www.edugrabs.com/3nf-third-normal-form/>

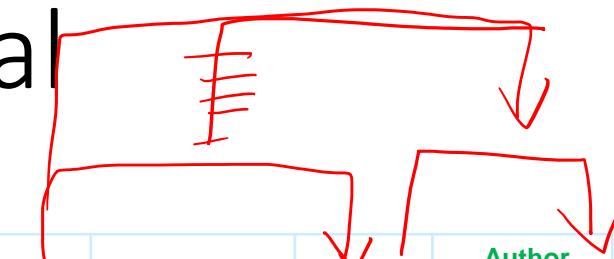
# Third Normal Form (3 NF)

- A **transitive dependency** is a functional dependency which holds by virtue of transitivity. A transitive dependency can occur only in a relation that has three or more attributes.
- Let A, B, and C designate three distinct attributes (or distinct collections of attributes) in the relation. Suppose all three of the following conditions hold:
  - $A \rightarrow B$
  - It is not the case that  $B \rightarrow A$
  - $B \rightarrow C$
- Then the functional dependency  $A \rightarrow C$  (which follows from 1 and 3 by the axiom of transitivity) is a transitive dependency



# Third Normal Form (3 NF)

- Example of **transitive dependency**
- The functional dependency  $\{Book\} \rightarrow \{\text{Author Nationality}\}$  applies; that is, if we know the book, we know the author's nationality. Furthermore:
  - $\{Book\} \rightarrow \{\text{Author}\}$
  - $\{\text{Author}\}$  does not  $\rightarrow \{Book\}$
  - $\{\text{Author}\} \rightarrow \{\text{Author Nationality}\}$
- Therefore  $\{Book\} \rightarrow \{\text{Author Nationality}\}$  is a transitive dependency.
- Transitive dependency occurred because a non-key attribute (**Author**) was determining another non-key attribute (**Author Nationality**).



| Book                                  | Genre                   | Author       | Author Nationality |
|---------------------------------------|-------------------------|--------------|--------------------|
| Twenty Thousand Leagues Under the Sea | Science Fiction         | Jules Verne  | French             |
| Journey to the Center of the Earth    | Science Fiction         | Jules Verne  | French             |
| Leaves of Grass                       | Poetry                  | Walt Whitman | American           |
| Anna Karenina                         | Literary Fiction        | Leo Tolstoy  | Russian            |
| A Confession                          | Religious Autobiography | Leo Tolstoy  | Russian            |

■ Example:

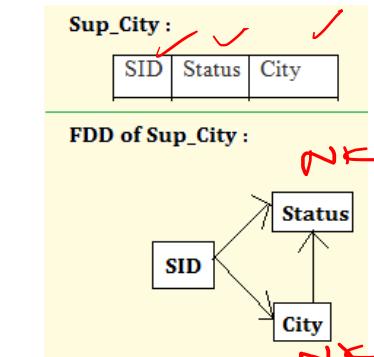
- **Sup\_City(SID, Status, City)** (already in 2NF)

| Sup_City: |        |        |
|-----------|--------|--------|
| SID       | Status | City   |
| S1        | 30     | Delhi  |
| S2        | 10     | Karnal |
| S3        | 40     | Rohtak |
| S4        | 30     | Delhi  |

**SID:** Primary Key

- Redundancy?
  - Status
- Anomaly?
  - Yes

# Third Normal Form (3 NF)



**Functional Dependencies:**  
 $SID \rightarrow Status$ ,  $SID \rightarrow City$   
 $City \rightarrow Status$

**Transitive Dependency :**  
 $SID \rightarrow Status$  {As  $SID \rightarrow City$  and  $City \rightarrow Status$ }

PPD

Retain the key in some decomposed relation

Post Normalization

| SC: |        |
|-----|--------|
| SID | City   |
| S1  | Delhi  |
| S2  | Karnal |
| S3  | Rohtak |
| S4  | Delhi  |

**SID:** Primary Key

| CS:    |        |
|--------|--------|
| City   | Status |
| Delhi  | 30     |
| Karnal | 10     |
| Rohtak | 40     |
| Delhi  |        |

**City:** Primary Key

$S \rightarrow City$   
 $C \rightarrow Status$

$R_1 \cap R_2 \rightarrow R_1$

Key in any of the decomposed table.

- The above two relations SC and CS are
1. Lossless Join
  2. **3NF**
  3. Dependency Preserving

# Third Normal Form (3 NF)

- Example
  - Relation *dept\_advisor*:
- ***dept\_advisor (s\_ID, i\_ID, dept\_name)***
- $F = \{s\_ID, \text{dept\_name} \rightarrow i\_ID, i\_ID \rightarrow \text{dept\_name}\}$
- Two candidate keys: ***s\_ID, dept\_name***, and ***i\_ID, s\_ID***
- *R* is in 3NF
  - ***s\_ID, dept\_name → i\_ID***
    - ***s\_ID, dept\_name*** is a superkey
  - ***i\_ID → dept\_name***
    - ***dept\_name*** is contained in a candidate key

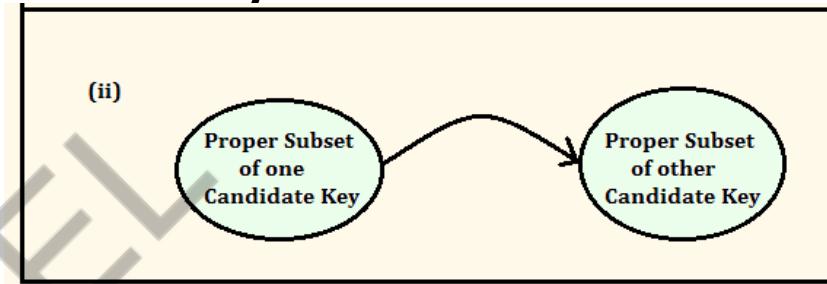
A relational schema **R** is in 3NF if for every FD  $X \rightarrow A$  associated with R either

- **A ⊥X** (i.e., the FD is trivial) or
- **X** is a superkey of **R** or
- **A** is part of some key (not just superkey!)

- A table is automatically in 3NF if one of the following hold :
  - (i) If relation consists of two attributes.
  - (ii) If 2NF table consists of only one non key attributes
- If  $X \rightarrow A$  is a dependency, then the table is in the 3NF, if one of the following conditions exists:
  - If X is a superkey
  - If X is a part of superkey
- If  $X \rightarrow A$  is a dependency, then the table is said to be NOT in 3NF if the following:
  - If X is a proper subset of some key (partial dependency)
  - If X is not a proper subset of key (non key)

# Third Normal Form (3 NF): Possible Redundancy

PPD



Source: <http://www.edugrabs.com/2nf-second-normal-form/>

## PPD

- Normal Forms
- **Decomposition to 3NF**
- Decomposition to BCNF

# DECOMPOSITION TO 3NF

# Testing for 3NF

- Optimization: Need to check only FDs in  $F$ , need not check all FDs in  $F^+$ .
- Use attribute closure to check for each dependency  $\alpha \rightarrow \beta$ , if  $\alpha$  is a superkey.
- If  $\alpha$  is not a superkey, we have to verify if each attribute in  $\beta$  is contained in a candidate key of  $R$ 
  - this test is rather more expensive, since it involve finding candidate keys
  - testing for 3NF has been shown to be NP-hard
  - Interestingly, decomposition into third normal form (described shortly) can be done in polynomial time

# 3NF Decomposition Algorithm

- Given: relation R, set F of functional dependencies
- Find: decomposition of R into a set of 3NF relation R<sub>i</sub>       $r_1, r_2 \dots r_n$
- Algorithm:
  1. Eliminate redundant FDs, resulting in a canonical cover F<sub>c</sub> of F
  2. Create a relation R<sub>i</sub> = XY for each FD  $X \rightarrow Y$  in F<sub>c</sub>
  3. If the key K of R does not occur in any relation R<sub>i</sub>, create one more relation R<sub>i</sub>=K

1  
6  
. 2  
9  
D  
a  
@  
S\_a  
i\_b  
l\_a  
b\_s  
ee  
r\_s  
sy  
c\_s  
ht  
ae  
tm  
zc  
, o  
Kn  
oc  
re  
tp  
ht  
as  
n\_  
d6  
S\_h  
u\_d  
di  
t  
ai  
r\_o  
s  
h  
a

# 3NF Decomposition Algorithm (Formal)

Let  $F_c$  be a canonical cover for  $F$ ;

$i := 0$ ;

**for each** functional dependency  $\alpha \rightarrow \beta$  in  $F_c$  **do**

**if** none of the schemas  $R_j$ ,  $1 \leq j \leq i$  contains  $\alpha \beta$

**then begin**

$i := i + 1$ ;

$R_i := \alpha \beta$

**end**

**if** none of the schemas  $R_j$ ,  $1 \leq j \leq i$  contains a candidate key for  $R$

**then begin**

$i := i + 1$ ;

$R_i :=$  any candidate key for  $R$ ;

**end**

/\* Optionally, remove redundant relations \*/

**repeat**

**if** any schema  $R_j$  is contained in another schema  $R_k$

**then** /\* delete  $R_j$  \*/

$R_j = R_j;;$

$i = i - 1$ ;

**return**  $(R_1, R_2, \dots, R_i)$

# 3NF Decomposition Algorithm

- Above algorithm ensures:
  - Each relation schema  $R_i$  is in 3NF
  - Decomposition is
    - Dependency preserving and
    - Lossless-join

# Example of 3NF Decomposition

- Relation schema:  
 $cust\_banker\_branch = (\underline{customer\_id}, \underline{employee\_id}, branch\_name, type)$
- The functional dependencies for this relation schema are:
  1.  $customer\_id, employee\_id \rightarrow branch\_name, type$
  2.  $employee\_id \rightarrow branch\_name$
  3.  $customer\_id, branch\_name \rightarrow employee\_id$
- We first compute a canonical cover
  - $branch\_name$  is extraneous in the r.h.s. of the 1<sup>st</sup> dependency
  - No other attribute is extraneous, so we get  $F_C =$   
 $customer\_id, employee\_id \rightarrow type$   
 $employee\_id \rightarrow branch\_name$   
 $customer\_id, branch\_name \rightarrow employee\_id$

# Example of 3NF Decomposition

- The **for** loop generates following 3NF schema:

- Observe that  $(customer\_id, employee\_id, type)$  contains a candidate key of the original schema, so no further relation schema needs be added
- At end of for loop, detect and delete schemas, such as  $(employee\_id, branch\_name)$ , which are subsets of other schemas

- result will not depend on the order in which FDs are considered

- The resultant simplified 3NF schema is:

$(customer\_id, employee\_id, type)$

$(customer\_id, branch\_name,$   
 $employee\_id)$

# Practice Problem for 3NF

## Decomposition: 1

PPD

- $R = ABCDEFGH$
- FDs = { $A \rightarrow B$ ,  $ABCD \rightarrow E$ ,  $EF \rightarrow GH$ ,  $ACDF \rightarrow EG$ }

Solution is given in the next slide (hidden from presentation – check after you have solved)

# Solution: Practice Problem for 3NF Decomposition: 1

PPD

- $R = ABCDEFGH$
- $FD = F = \{A \rightarrow B, ABCD \rightarrow E, EF \rightarrow GH, ACDF \rightarrow EG\}$
- Compute Canonical Cover ( $F_c$ )
  - Make RHS a single attribute:  $\{A \rightarrow B, ABCD \rightarrow E, EF \rightarrow G, EF \rightarrow H, ACDF \rightarrow E, ACDF \rightarrow G\}$
  - Minimize LHS:  $ACD \rightarrow E$  instead of  $ABCD \rightarrow E$
  - Eliminate redundant FDs
    - Can  $ACDF \rightarrow G$  be removed?
    - Can  $ACDF \rightarrow E$  be removed?
  - $F_c = \{A \rightarrow B, ACD \rightarrow E, EF \rightarrow G, EF \rightarrow H\}$
- Superkey =  $ACDF$
- 3NF Decomposition =  $\{AB, ACDE, EFG, EFH\} \cup \{ACDF\}$

 $R_1, R_2, R_3, R_4, R_5$ 

$$\begin{aligned} A \rightarrow B &\rightarrow R_1(A \ B) \\ ACD \rightarrow E &\rightarrow R_2(A \ C \ D \ E) \\ EF \rightarrow G &\rightarrow R_3(E \ F \ G) \\ EF \rightarrow H &\rightarrow R_4(E \ F \ H) \\ ACDF &= R_5(A \ C \ D \ F) \end{aligned}$$

This is a hidden slide giving solution. Check only after you have solved)

# Practice Problem for 3NF

## Decomposition: 2

PPD

- $R = CSJDPQV$
- FDs = { $C \rightarrow CSJDPQV$ ,  $SD \rightarrow P$ ,  $JP \rightarrow C, J \rightarrow S$ }

Solution is given in the next slide (hidden from presentation – check after you have solved)

1

# Solution: Practice Problem for 3NF Decomposition: 2

- $R = CSJDPQV$
- FDs =  $F = \{C \rightarrow CSJDPQV, SD \rightarrow P, JP \rightarrow C, J \rightarrow S\}$
- Compute Canonical Cover ( $F_c$ )
  - Minimal cover:  $\{C \rightarrow J, C \rightarrow D, C \rightarrow Q, C \rightarrow V, JP \rightarrow C, J \rightarrow S, SD \rightarrow P\}$  (Combine LHS)
  - $F_c = \{C \rightarrow JDQV, JP \rightarrow C, J \rightarrow S, SD \rightarrow P\}$
- Superkey = C
- 3NF Decomposition = {CJDQV, JPC, JS, SDP}

This is a hidden slide giving solution. Check only after you have solved)

- Normal Forms
- Decomposition to 3NF
- **Decomposition to BCNF**

## DECOMPOSITION TO BCNF

# Testing for BCNF

Boyce-Codd NF

- To check if a non-trivial dependency  $\alpha \rightarrow \beta$  causes a violation of BCNF
  - 1. compute  $\alpha^+$  (the attribute closure of  $\alpha$ ), and
  - 2. verify that it includes all attributes of  $R$ , that is, it is a superkey of  $R$ .
- **Simplified test:** To check if a relation schema  $R$  is in BCNF, it suffices to check only the dependencies in the given set  $F$  for violation of BCNF, rather than checking all dependencies in  $F^+$ .
  - If none of the dependencies in  $F$  causes a violation of BCNF, then none of the dependencies in  $F^+$  will cause a violation of BCNF either.
- However, **simplified test using only  $F$  is incorrect when testing a relation in a decomposition of  $R$** 
  - Consider  $R = (A, B, C, D, E)$ , with  $F = \{ A \rightarrow B, BC \rightarrow D \}$ 
    - Decompose  $R$  into  $R_1 = (A, B)$  and  $R_2 = (A, C, D, E)$
    - Neither of the dependencies in  $F$  contain only attributes from  $(A, C, D, E)$  so we might be misled into thinking  $R_2$  satisfies BCNF.
    - In fact, dependency  $AC \rightarrow D$  in  $F^+$  shows  $R_2$  is not in BCNF.

# Testing Decomposition for BCNF

- To check if a relation  $R_i$  in a decomposition of  $R$  is in BCNF,
  - Either test  $R_i$  for BCNF with respect to the **restriction** of  $F$  to  $R_i$  (that is, all FDs in  $F^+$  that contain only attributes from  $R_i$ )
  - or use the original set of dependencies  $F$  that hold on  $R$ , but with the following test:
    - for every set of attributes  $\alpha \subseteq R_i$ , check that  $\alpha^+$  (the attribute closure of  $\alpha$ ) either includes no attribute of  $R_i - \alpha$ , or includes all attributes of  $R_i$ .
    - If the condition is violated by some  $\alpha \rightarrow \beta$  in  $F$ , the dependency
$$\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i$$
can be shown to hold on  $R_i$ , and  $R_i$  violates BCNF.
    - We use above dependency to decompose  $R_i$

# BCNF Decomposition Algorithm

PPD

1. For all dependencies  $A \rightarrow B$  in  $F^+$ , check if A is a superkey
  - By using attribute closure
2. If not, then
  - Choose a dependency in  $F^+$  that breaks the BCNF rules, say  $A \rightarrow B$
  - Create  $R1 = A B$
  - Create  $R2 = (R - B)$
  - Note that:  $R1 \cap R2 = A$  and  $A \rightarrow AB (= R1)$ , so this is lossless decomposition
3. Repeat for  $R1$ , and  $R2$ 
  - By defining  $F1^+$  to be all dependencies in  $F$  that contain only attributes in  $R1$
  - Similarly  $F2^+$

# BCNF Decomposition Algorithm

```
result := {R};
done := false;
compute F^+ ;
while (not done) do
 if (there is a schema R_i in result that is not in BCNF)
 then begin
 let $\alpha \rightarrow \beta$ be a nontrivial functional dependency that
 holds on R_i such that $\alpha \rightarrow R_i$ is not in F^+ ,
 and $\alpha \cap \beta = \emptyset$;
 result := (result - R_i) \cup ($R_i - \beta$) \cup (α, β);
 end
 else done := true;
```

Note: each  $R_i$  is in BCNF, and decomposition is lossless-join.

# Example of BCNF Decomposition

- $R = (A, B, C)$   
 $F = \{A \rightarrow B$   
 $\quad \quad B \rightarrow C\}$   
Key = {A}
- R is not in BCNF ( $B \rightarrow C$  but B is not superkey)
- Decomposition
  - $R_1 = (B, C)$
  - $R_2 = (A, B)$

# Example of BCNF Decomposition

- *class (course\_id, title, dept\_name, credits, sec\_id, semester, year, building, room\_number, capacity, time\_slot\_id)*
- Functional dependencies:
  - $course\_id \rightarrow title, dept\_name, credits$
  - $building, room\_number \rightarrow capacity$
  - $course\_id, sec\_id, semester, year \rightarrow building, room\_number, time\_slot\_id$
- A candidate key  $\{course\_id, sec\_id, semester, year\}$ .
- BCNF Decomposition:
  - $course\_id \rightarrow title, dept\_name, credits$  holds
    - but  $course\_id$  is not a superkey.
  - We replace *class* by:
    - *course(course\_id, title, dept\_name, credits)*
    - *class-1 (course\_id, sec\_id, semester, year, building, room\_number, capacity, time\_slot\_id)*

# BCNF

## Decomposition

### (Cont.)

- course is in BCNF
  - How do we know this?
- $building, room\_number \rightarrow capacity$  holds on  $class-1(course\_id, sec\_id, semester, year, building, room\_number, capacity, time\_slot\_id)$ 
  - but  $\{building, room\_number\}$  is not a superkey for  $class-1$ .
  - We replace  $class-1$  by:
    - $classroom (building, room\_number, capacity)$
    - $section (course\_id, sec\_id, semester, year, building, room\_number, time\_slot\_id)$
- $classroom$  and  $section$  are in BCNF.

# BCNF and Dependency Preservation

- It is not always possible to get a BCNF decomposition that is dependency preserving
- $R = (J, K, L)$   
 $F = \{JK \rightarrow L$   
 $L \rightarrow K\}$   
Two candidate keys =  $JK$  and  $JL$
- $R$  is not in BCNF
- Any decomposition of  $R$  will fail to preserve  
 $JK \rightarrow L$   
This implies that testing for  $JK \rightarrow L$  requires a join

# Practice Problem for BCNF Decomposition

- R = ABCDE. F = {A → B, BC → D}
- R = ABCDE. F = {A → B, BC → D}
- R = ABCDEH. F = {A → BC, E → HA}
- R = CSJDPQV. F = {C → CSJDPQV, SD → P, JP → C, J → S}
- R = ABCD. F = {C → D, C → A, B → C}

# Comparison of BCNF and 3NF

- It is always possible to decompose a relation into a set of relations that are in 3NF such that:
  - the decomposition is lossless
  - the dependencies are preserved
- It is always possible to decompose a relation into a set of relations that are in BCNF such that:
  - the decomposition is lossless
  - it may not be possible to preserve dependencies.

| S# | 3NF                                                                                                  | BCNF                                                            |
|----|------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------|
| 1. | It concentrates on Primary Key                                                                       | It concentrates on Candidate Key.                               |
| 2. | Redundancy is high as compared to BCNF                                                               | 0% redundancy                                                   |
| 3. | It may preserve all the dependencies                                                                 | It may not preserve the dependencies.                           |
| 4. | A dependency $X \rightarrow Y$ is allowed in 3NF if $X$ is a super key or $Y$ is a part of some key. | A dependency $X \rightarrow Y$ is allowed if $X$ is a super key |

Ex:  $R(N, D, P, S, A)$

fd1:  $N \rightarrow D$

fd2:  $S \rightarrow A$

fd3:  $(NS) \rightarrow P$

Q) what is the highest possible normal form seen in the given example  
+ check get the relations to (2NF) 3NF | BCNF ?

i) check for 2NF , Partial dependency

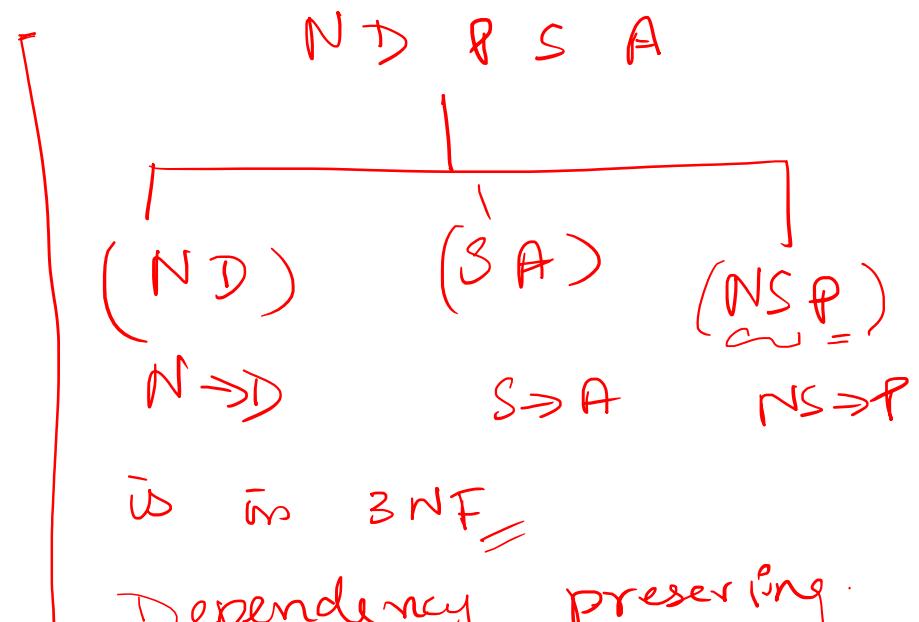
Candidate Key =  $(N, S)$

fd1 = violations of 2NF | not in 2NF

fd2 = not in 2NF

fd3 = is in 2NF

$R$  is not in 2NF but in 1NF



Dependency preserving decomposition

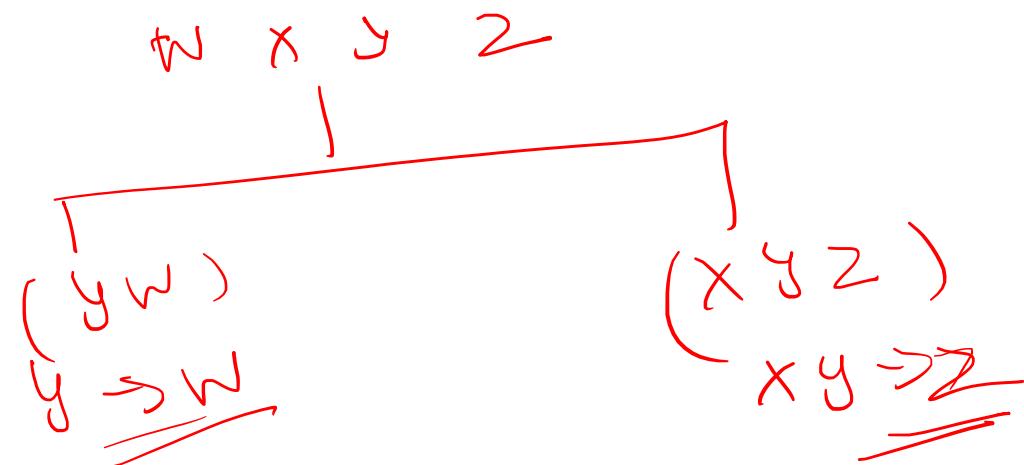
Ex:-  $R(w, x, y, z)$

$$xy \Rightarrow z$$

$$y \rightarrow w$$

$$cx = xy$$

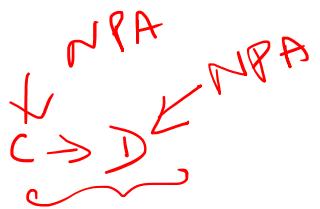
Is it in 2NF : No ; fd2 is violating



This is in 3NF and dependency preserving.

Ex3:-  $R(A B C D)$

$$A \Rightarrow BC$$



$$A \rightarrow AB$$

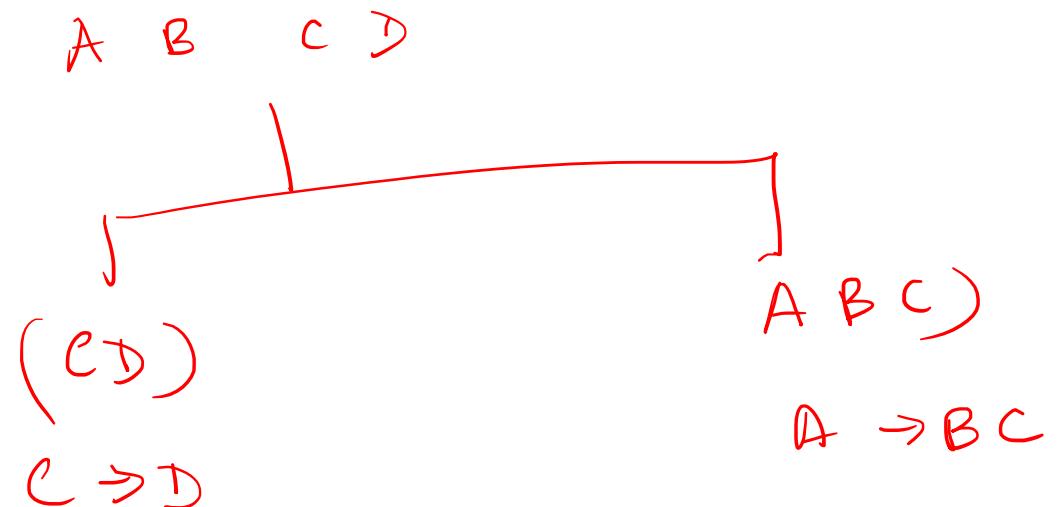
$$DB \rightarrow AB$$

Is it in 2NF : Yes

Is it in 3NF : No

$$CF = A$$

fd2 violates 3NF



$F_C$

It is in 3NF and dependency preserving

$\Rightarrow$  R( C S SD P Q V )

fd1 : C  $\rightarrow$  CS SD P Q V

fd2 : SD  $\rightarrow$  P

fd3 : JP  $\rightarrow$  Q

fd4 : J  $\rightarrow$  S

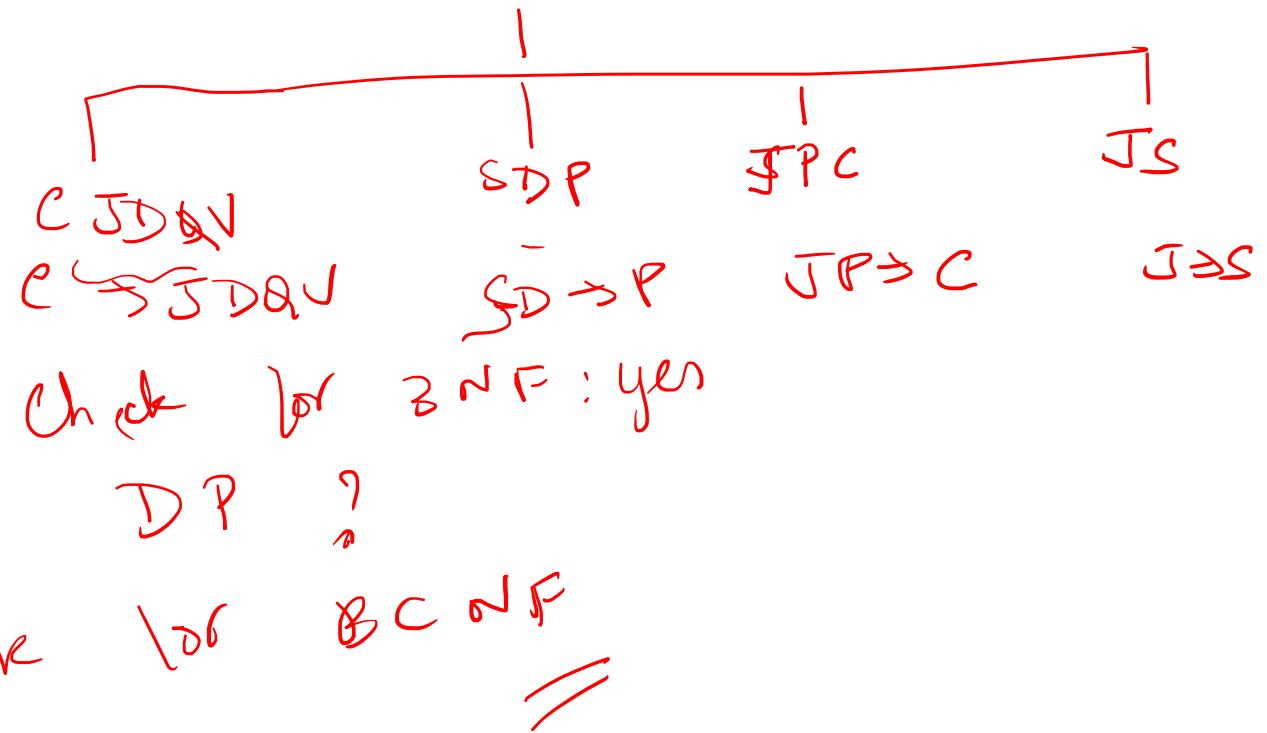
Sol: ek is, C

$\downarrow$  it is in 2NF : yes

is it in 3NF : no

Find Canonical cover ; For each fd in CC make a relation. If key is not in any decomposed table then create new relation having key

$$\begin{aligned}
 F_C = & C \rightarrow JDQV \\
 & SD \rightarrow P \\
 & JP \rightarrow C \\
 & J \rightarrow S
 \end{aligned}$$



Ex:-  $R(A \ B \ C \ D \ E \ F \ G_1 \ H)$

workout for  
students

$A \rightarrow B$

$ABCD \rightarrow E$

$EF \rightarrow GH$

$ACDF \rightarrow EG_1$

Check for 2NF | 3NF | BCNF  
Get the relations in 3NF and BCNF

- **Multivalued Dependencies**
- Decomposition to 4NF
- Database-Design Process
- Modeling Temporal Data

## MULTIVALUED DEPENDENCY

→ → not allow b & m vD

PPD

# Multivalued Dependency

## ■ *Persons(Man, Phones, Dog\_Like)*

| Person :  |           |              | Meaning of the tuples                                   |
|-----------|-----------|--------------|---------------------------------------------------------|
| Man(M)    | Phones(P) | Dogs_Like(D) | Man M have phones P, and likes the dogs D.              |
| M1        | P1/P2     | D1/D2        | M1 have phones P1 and P2, and likes the dogs D1 and D2. |
| M2        | P3        | D2           | M2 have phones P3, and likes the dog D2.                |
| Key : MPD |           |              |                                                         |

There are no non trivial FDs because all attributes are combined forming Candidate Key i.e. MDP. In the above relation, two multivalued dependencies exists –

- **Man →→ Phones**
- **Man →→ Dogs\_Like**

A man's phone are independent of the dogs they like. But after converting the above relation in Single Valued Attribute, each of a man's phones appears with each of the dogs they like in all combinations.

Source: <http://www.edugrabs.com/multivalued-dependency-mvd/>

## Post 1NF Normalization

| Man(M) | Phones(P) | Dogs_Likes(D) |
|--------|-----------|---------------|
| M1     | P1        | D1            |
| M1     | P2        | D2            |
| M2     | P3        | D2            |
| M1     | P1        | D2            |
| M1     | P2        | D1            |

- Multivalued dependency occurs when two attributes in a table are independent of each other but, both depend on a third attribute.

$(A) \Rightarrow B$  ✓

$A \Rightarrow C$  //

but  $B$  and  $C$  are not connected.

# Multivalued Dependency

- If two or more independent relations are kept in a single relation, then Multivalued Dependency is possible. For example, Let there are two relations :
  - ***Student(SID, Sname)*** where  $(SID \rightarrow Sname)$
  - ***Course(CID, Cname)*** where  $(CID \rightarrow Cname)$
- There is no relation defined between Student and Course. If we kept them in a single relation named ***Student\_Course***, then MVD will exists because of *minCardinality* min
- If two or more MVDs exist in a relation, then while converting into SVAs, MVD exists.

| Student: |       |
|----------|-------|
| SID      | Sname |
| S1       | A     |
| S2       | B     |

| Course: |       |
|---------|-------|
| CID     | Cname |
| C1      | C     |
| C2      | B     |

| SID | Sname | CID | Cname |
|-----|-------|-----|-------|
| S1  | A     | C1  | C     |
| S1  | A     | C2  | B     |
| S2  | B     | C1  | C     |
| S2  | B     | C2  | B     |

2 MVDs exist:

1.  $SID \rightarrow\rightarrow CID$
2.  $SID \rightarrow\rightarrow Cname$

# Multivalued Dependencies (MVDs)

PPD

- Let  $R$  be a relation schema and let  $\alpha \subseteq R$  and  $\beta \subseteq R$ . The **multivalued dependency**

$$\alpha \rightarrow\!\!\!\rightarrow \beta$$

holds on  $R$  if in any legal relation  $r(R)$ , for all pairs of tuples  $t_1$  and  $t_2$  in  $r$  such that  $t_1[\alpha] = t_2[\alpha]$ , there exist tuples  $t_3$  and  $t_4$  in  $r$  such that:

$$\begin{aligned}
 t_1[\alpha] &= t_2[\alpha] = t_3[\alpha] = t_4[\alpha] \\
 t_3[\beta] &= t_1[\beta] \\
 t_3[R - \beta] &= t_2[R - \beta] \\
 t_4[\beta] &= t_2[\beta] \\
 t_4[R - \beta] &= t_1[R - \beta]
 \end{aligned}$$

Example: A relation of university courses, the books recommended for the course, and the lecturers who will be teaching the course:

- $\text{course} \rightarrow\!\!\!\rightarrow \text{book}$
- $\text{course} \rightarrow\!\!\!\rightarrow \text{lecturer}$

Test:  $\text{course} \rightarrow\!\!\!\rightarrow \text{book}$

| Course | Book           | Lecturer    | Tuples |
|--------|----------------|-------------|--------|
| AHA ✓  | Silberschatz ✓ | John D ✓    | t1     |
| AHA ✓  | Nederpelt      | William M ✓ | t2     |
| AHA ✓  | Silberschatz ✓ | William M ✓ | t3     |
| AHA ✓  | Nederpelt ✓    | John D ✓    | t4     |
| AHA    | Silberschatz   | Christian G |        |
| AHA    | Nederpelt      | Christian G |        |
| OSO    | Silberschatz   | John D      |        |
| OSO    | Silberschatz   | William M   |        |

# Example

- Let  $R$  be a relation schema with a set of attributes that are partitioned into 3 nonempty subsets.  
 $Y, Z, W$
- We say that  $Y \rightarrow\!\!\rightarrow Z$  ( $Y$  **multidetermines**  $Z$ ) if and only if for all possible relations  $r(R)$   
 $< y_1, z_1, w_1 > \in r$  and  $< y_1, z_2, w_1 > \in r$   
then  
 $< y_1, z_1, w_2 > \in r$  and  $< y_1, z_2, w_2 > \in r$
- Note that since the behavior of  $Z$  and  $W$  are identical it follows that  
 $Y \rightarrow\!\!\rightarrow Z$  if  $Y \rightarrow\!\!\rightarrow W$

# Example (Cont.)

- In our example:

$ID \rightarrow\rightarrow child\_name$

$ID \rightarrow\rightarrow phone\_number$

- The above formal definition is supposed to formalize the notion that given a particular value of  $Y$  ( $ID$ ) it has associated with it a set of values of  $Z$  ( $child\_name$ ) and a set of values of  $W$  ( $phone\_number$ ), and these two sets are in some sense independent of each other.
- Note:
  - If  $Y \rightarrow Z$  then  $Y \rightarrow\rightarrow Z$
  - Indeed we have (in above notation)  $Z_1 = Z_2$   
The claim follows.

# Use of Multivalued Dependencies

- We use multivalued dependencies in two ways:
  1. To test relations to **determine** whether they are legal under a given set of functional and multivalued dependencies
  2. To specify **constraints** on the set of legal relations. We shall thus concern ourselves *only* with relations that satisfy a given set of functional and multivalued dependencies.
- If a relation  $r$  fails to satisfy a given multivalued dependency, we can construct a relation  $r'$  that does satisfy the multivalued dependency by adding tuples to  $r$ .



# Theory of MVDs

PPD

|    | Name                   | Rule                                                                                                                                                   |
|----|------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| C- | <b>Complementation</b> | : If $X \rightarrow\rightarrow Y$ , then $X \rightarrow\rightarrow \{R - (X \cup Y)\}$ .                                                               |
| A- | <b>Augmentation ✓</b>  | : If $X \rightarrow\rightarrow Y$ and $W \supseteq Z$ , then $WX \rightarrow\rightarrow YZ$ .                                                          |
| T- | <b>Transitivity ✓</b>  | : If $X \rightarrow\rightarrow Y$ and $Y \rightarrow\rightarrow Z$ , then $X \rightarrow\rightarrow (Z - Y)$ .                                         |
|    | <b>Replication</b>     | : If $X \rightarrow Y$ , then $X \rightarrow\rightarrow Y$ but the reverse is not true.                                                                |
|    | <b>Coalescence</b>     | : If $X \rightarrow\rightarrow Y$ and there is a $W$ such that $W \cap Y$ is empty, $W \rightarrow Z$ , and $Y \supseteq Z$ , then $X \rightarrow Z$ . |

- A MVD  $X \rightarrow\rightarrow Y$  in  $R$  is called a trivial MVD if
  - $Y$  is a subset of  $X$  ( $X \supseteq Y$ ) or
  - $X \cup Y = R$ . Otherwise, it is a non trivial MVD and we have to repeat values redundantly in the tuples.

Source: <http://www.edugrabs.com/multivalued-dependency-mvd/>

# Theory of MVDs

- From the definition of multivalued dependency, we can derive the following rule:
  - If  $\alpha \rightarrow \beta$ , then  $\alpha \rightarrow\rightarrow \beta$

That is, every functional dependency is also a multivalued dependency
- The **closure**  $D^+$  of  $D$  is the set of all functional and multivalued dependencies logically implied by  $D$ .
  - We can compute  $D^+$  from  $D$ , using the formal definitions of functional dependencies and multivalued dependencies.
  - We can manage with such reasoning for very simple multivalued dependencies, which seem to be most common in practice
  - For complex dependencies, it is better to reason about sets of dependencies using a system of inference rules



•That's all.

folks!!



# ADVANCED SQL: TRIGGERS & ASSERTIONS

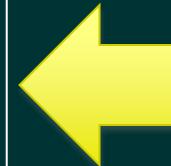
1

# Triggers

# Triggers: Introduction

- The application constraints need to be captured inside the database
- **Some constraints can be captured by:**
  - Primary Keys, Foreign Keys, Unique, Not NULL, and domain constraints

```
CREATE TABLE Students
 (sid: CHAR(20),
 name: CHAR(20) NOT NULL,
 login: CHAR(10),
 age: INTEGER,
 gpa: REAL Default 0,
 Constraint pk Primary Key (sid),
 Constraint u1 Unique (login),
 Constraint gpaMax check (gpa <= 4.0));
```



These constraints are defined in ***CREATE TABLE*** or ***ALTER TABLE***

# Triggers: Introduction

- **Other application constraints are more complex**
  - Need for assertions and triggers
- **Examples:**
  - Sum of loans taken by a customer does not exceed 100,000
  - Student cannot take the same course after getting a pass grade in it
  - Age field is derived automatically from the Date-of-Birth field

# Triggers

- A procedure that runs automatically when a certain **event** occurs in the DBMS
- **The procedure performs some actions, e.g.,**
  - Check certain values
  - Fill in some values
  - Inserts/deletes/updates other records
  - Check that some business constraints are satisfied
  - Commit (approve the transaction) or roll back (cancel the transaction)

# Trigger Components

- **Three components**

- **Event:** When this event happens, the trigger is activated
- **Condition (optional):** If the condition is true, the trigger executes, otherwise skipped
- **Action:** The actions performed by the trigger

- **Semantics**

- When the **Event** occurs and **Condition** is true, execute the **Action**

# Trigger: Events

- **Three event types**
  - Insert
  - Update
  - Delete
- **Two triggering times**
  - Before the event
  - After the event
- **Two granularities**
  - Execute for each row
  - Execute for each statement

# 1) Trigger: Event

**Create Trigger <name>**  
**Before|After    Insert|Update|Delete    ON <tablename>**

....

Trigger name  
That is the event

- Example

**Create Trigger ABC**  
**Before Insert On Students**

....



This trigger is activated when an insert statement is issued, but before the new record is inserted

**Create Trigger XYZ**  
**After Update On Students**

....



This trigger is activated when an update statement is issued and after the update is executed

# Granularity of Event

- A single SQL statement may update, delete, or insert many records at the same time
  - E.g., **Update student set gpa = gpa x 0.8;**
- **Does the trigger execute for each updated or deleted record, or once for the entire statement ?**
  - **We define such granularity**

```
Create Trigger <name>
Before| After Insert| Update| Delete ← This is the event

For Each Row | For Each Statement ← This is the granularity
....
```

# Example: Granularity of Event

**Create Trigger XYZ**

**After Update ON <tablename>**

**For each statement**

....

This trigger is activated once (per UPDATE statement) after all records are updated

**Create Trigger XYZ**

**Before Delete ON <tablename>**

**For each row**

....

This trigger is activated before deleting each record

## 2) Trigger: Condition

- This component is **optional**

**Create Trigger <name>**

**Before| After      Insert| Update| Delete On <tableName>**

**For Each Row | For Each Statement**

**When <condition>**

That is the condition

...

**If the employee salary > 150,000 then some actions will be taken**

**Create Trigger EmpSal**

**After Insert or Update On Employee**

**For Each Row**

**When (new.salary >150,000)**

...

### 3) Trigger: Action

- **Action depends on what you want to do, e.g.:**

- Check certain values
- Fill in some values
- Inserts/deletes/updates other records
- Check that some business constraints are satisfied
- Commit (approve the transaction) or roll back (cancel the transaction)

- **In the action, you may want to reference:**

- The new values of inserted or updated records (**:new**)
- The old values of deleted or updated records (**:old**)

# Trigger: Referencing Values

- **In the action, you may want to reference:**

- The new values of inserted or updated records (**:new**)
- The old values of deleted or updated records (**:old**)

**Trigger  
body**

```
Create Trigger EmpSal
After Insert or Update On Employee
For Each Row
When (new.salary > 150,000)
Begin
 if (:new.salary < 100,000) ...
End;
```

Inside “When”, the “new” and “old” should not have “:”

Inside the trigger body, they should have “:”

# Trigger: Referencing Values (Cont'd)

- **Insert Event**

- Has only :new defined

- **Delete Event**

- Has only :old defined

- **Update Event**

- Has both :new and :old defined

- **Before triggering (for insert/update)**

- Can update the values in :new
  - Changing :old values does not make sense

- **After triggering**

- Should not change :new because the event is already done

# Example 1

If the employee salary increased by more than 10%, make sure the 'rank' field is not empty and its value has changed, otherwise reject the update

```
If the trigger exists, then drop it first
Create or Replace Trigger EmpSal
Before Update On Employee
For Each Row
Begin
 IF (:new.salary > (:old.salary * 1.1)) Then
 IF (:new.rank is null or :new.rank = :old.rank) Then
 RAISE_APPLICATION_ERROR (-20004, 'rank field not
correct');
 End IF;
 End IF;
End;
/ Make sure to have the "/" to run the command
```

## Example 2

If the employee salary increased by more than 10%, then increment the rank field by 1.

In the case of **Update** event only, we can specify which columns

**Create or Replace Trigger** *EmpSal*

**Before Update Of salary On Employee**

**For Each Row**

**Begin**

IF (:new.salary > (:old.salary \* 1.1)) Then

:new.rank := :old.rank + 1;

End IF;

**End;**

/

We changed the new value of **rank** field

The assignment operator has ":"

# Example 3: Using Temp Variable

If the newly inserted record in employee has null hireDate field, fill it in with the current date

```
Create Trigger EmpDate
Before Insert On Employee
For Each Row
Declare
 temp date;
Begin
 Select sysdate into temp from dual;
 IF (:new.hireDate is null) Then
 :new.hireDate := temp;
 End If;
End;
/
```

Since we need to change values, then it should be “Before” event

Declare section to define variables

Oracle way to select the current date

Updating the new value of hireDate before inserting it

# Example 4: Maintenance of Derived Attributes

Keep the bonus attribute in Employee table always 3% of the salary attribute

**Create Trigger** *EmpBonus*

**Before Insert Or Update On**

*Employee*

**For Each Row**

**Begin**

`:new.bonus := :new.salary * 0.03;`

**End;**

`/`

Indicate two events at the same time

The bonus value is always computed automatically

# Row-Level vs. Statement-Level Triggers

- **Example:** *Update emp set salary = 1.1 \* salary;*

- Changes many rows (records)

- **Row-level triggers**

- Check individual values and can update them
  - Have access to **:new** and **:old** vectors

- **Statement-level triggers**

- Do not have access to **:new** or **:old** vectors (only for row-level)
  - Execute once for the entire statement regardless how many records are affected
  - Used for verification before or after the statement

# Example 5: Statement-level Trigger

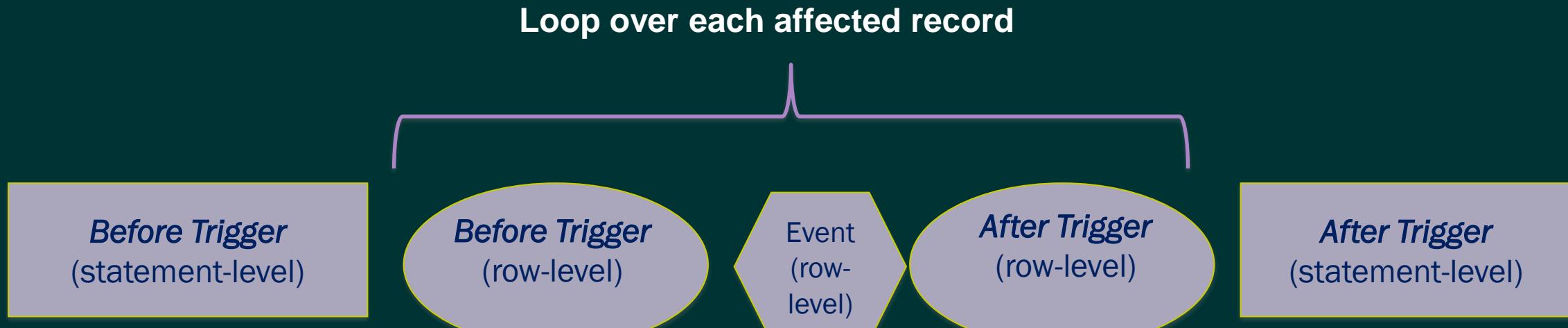
Store the count of employees having salary > 100,000 in table R

```
Create Trigger EmpBonus
After Insert Or Update of salary Or Delete On Employee
For Each Statement
Begin
 delete from R;
 insert into R(cnt) Select count(*) from employee where salary > 100,000;
End;
/
```

Indicate three events at the same time

Delete the existing record in R, and then insert the new count.

# Order Of Trigger Firing



# Some Other Operations

- Dropping Trigger

**SQL> Create Trigger <trigger name>;**

- If creating trigger with errors

**SQL > Show errors;**

# Assertions

# Assertions

- An expression that should be always true
- When created, the expression must be true
- DBMS checks the assertion after any change that may violate the expression

```
create assertion <assertion-name> check <predicate>
```

Must return True or False



# Example 1

**Sum of loans taken by a customer does not exceed 100,000**

- *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)*

Must return True or False  
(not a relation)

## Create Assertion SumLoans Check

( 100,000 >= ALL

Select Sum(amount)

From borrower B , loan L

Where B.loan\_number = L.loan\_number

Group By customer\_name );

# Example 2

- *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)*

**Number of accounts for each customer in a given branch is at most two**

**Create Assertion** NumAccounts **Check**

( 2 >= ALL

Select count(\*)

From account A , depositor D

Where A.account\_number = D.account\_number

Group By customer\_name, branch\_name );

# Example 3

- *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)*

**Customer city is always not null**

**Create Assertion CityCheck Check**

```
(NOT EXISTS (
 Select *
 From customer
 Where customer city is null));
```

# Assertions vs. Triggers

- Assertions do not modify the data, they only check certain conditions
- Triggers are more powerful because they can check conditions and also modify the data
- Assertions are not linked to specific tables in the database and not linked to specific events
- Triggers are linked to specific tables and specific events
- All assertions can be implemented as triggers (one or more)
- Not all triggers can be implemented as assertions
- Oracle does not have assertions

# Example: Trigger vs. Assertion

All new customers opening an account must have opening balance  $\geq \$100$ . However, once the account is opened their balance can fall below that amount.

We need triggers, assertions cannot be used



Trigger Event: Before Insert

```
Create Trigger OpeningBal
Before Insert On Customer
For Each Row
Begin
 IF (:new.balance is null or :new.balance < 100) Then
 RAISE_APPLICATION_ERROR(-20004, 'Balance should be $\geq \$100$ ');
 End IF;
End;
```