

# Table of Contents

---

5. ER Model 2023 DSE-Complete	8
1 ECDACoTF_F6q4k6_0	8
1.1 Slide 1	8
1.2 Slide 2	9
1.3 ER-Model	10
1.4 Entity	11
1.5 Entity Sets	12
1.6 Entity Sets- instructor and student	13
1.7 Relationship Sets	14
1.8 Relationship Sets	15
1.9 Relationship Sets (Cont.)	16
1.10 E-R Diagrams	17
1.11 Relationship Sets (Cont.)	18
1.12 Relationship Sets with Attributes	19
1.13 Recursive Relationship set	20
1.14 Recursive Relationship..	21
1.15 Recursive Relationship when converted into schema, it looks as below-	22
1.16 Attributes	23
1.17 Composite Attributes	24
1.18 Slide 18	25
1.19 Slide 19	26
1.20 Similar to UML- Class Diagram	27
1.21 Slide 21	28
1.22 Mapping Cardinality Constraints	29
1.23 Mapping Cardinalities	30
1.24 Mapping Cardinalities	31
1.25 Cardinality Constraints	32
1.26 One-to-One Relationship	33
1.27 Example one-one	34
1.28 Slide 28	35
1.29 Example one- Many	36
1.30 Slide 30	37
1.31 Example Many-one	38
1.32 Slide 32	39
1.33 Example Many - Many	40
1.34 Example- Design ER Model	41
1.35 Slide 35	42
1.36 Slide 36	43
1.37 Slide 37	44
1.38 Slide 38	45
1.39 Notation for Expressing More Complex Constraints	46
1.40 Notation for Expressing More Complex Constraints	47
1.41 Slide 41	48

1.42	Slide 42	49
1.43	Slide 43	50
1.44	Slide 44	51
1.45	Slide 45	52
1.46	Slide 46	53
1.47	Redundant Attributes	54
1.48	Keys	55
1.49	Keys for Relationship Sets	56
1.50	Keys for Relationship Sets	57
1.51	Keys for Relationship Sets having Attributes	58
1.52	Slide 52	59
1.53	Slide 53	60
1.54	Slide 54	61
1.55	Slide 55	62
1.56	Slide 56	63
1.57	Slide 57	64
1.58	Information to be fetched from Requirements	65
1.59	Slide 59	66
1.60	Slide 60	67
1.61	Slide 61	68
1.62	Slide 62	69
1.63	Slide 63	70
1.64	Slide 64	71
1.65	Reduction to Relation Schemas	72
1.66	Representing Entity Sets With Simple Attributes	73
1.67	Slide 67	74
1.68	Representing Relationship Sets Many-Many	75
1.69	Representing Relationship sets- Many-1/1-Many	76
1.70	Representing Relationship sets- 1 to 1	77
1.71	Slide 71	78
1.72	Slide 72	79
1.73	Slide 73	80
1.74	Slide 74	81
1.75	Slide 75	82
1.76	Slide 76	83
1.77	Slide 77	84
1.78	Slide 78	85
1.79	Slide 79	86
1.80	Representing Composite Attributes in Schema	87
1.81	Representing Multivalued Attributes in Schema	88
1.82	Extended E-R Features	89
1.83	Extended E-R Features: Specialization	90
1.84	Specialization / Generalization Example	91
1.85	Design Constraints on a Specialization/ Generalization	92
1.86	Design Constraints on a Specialization/Generalization (Cont.)	93

1.87	Representing Specialization via Schemas	94
1.88	Representing Specialization as Schemas (Cont.)	95
1.89	Slide 89	96
1.90	Slide 90	97
1.91	Slide 91	98
1.92	Slide 92	99
1.93	Slide 93	100
1.94	Summary of Symbols Used in E-R Notation	101
1.95	Symbols Used in E-R Notation (Cont.)	102
1.96	Alternative ER Notations	103
1.97	Alternative ER Notations	104
1.98	Slide 98	105
1.99	Slide 99	106
1.100	Slide 100	107
6.	PL_SQL Oracle DSE 2023	108
1	1890CoTF_K9qsxa_0	108
1.1	PL/SQL	108
1.2	What is PL/SQL	109
1.3	Why PL/SQL	110
1.4	PL/SQL BLOCK STRUCTURE	111
1.5	PL/SQL Block Types	112
1.6	PL/SQL Variable Types	113
1.7	DECLARE	114
1.8	PL/SQL- Assignment	115
1.9	DBMS_OUTPUT.PUT_LINE()	116
1.10	PL/SQL FIRST PROGRAM	117
1.11	PL/SQL Sample Program	118
1.12	PL/SQL sample program	119
1.13	Retrieving Column values into variables	120
1.14	Slide 14	121
1.15	SELECT.. INTO..	122
1.16	%TYPE	123
1.17	%ROWTYPE	124
1.18	%ROWTYPE	125
1.19	Example:	126
1.20	COMMON IN-BUILT STRING FUNCTIONS	127
1.21	COMMON IN-BUILT NUMERIC FUNCTIONS	128
1.22	Conditional logic	129
1.23	IF-THEN-ELSIF Statements	130
1.24	CASE.. WHEN Statement	131
1.25	CASE.. WHEN- Example	132
1.26	Loops: Simple Loop	133
1.27	Loops: FOR Loop	134
1.28	Example-Loops: FOR Loop	135
1.29	Example-Loops: FOR Loop ( REVERSE)	136

1.30 Example-Loops: WHILE Loop	137
1.31 Cursors	138
1.32 CURSORS	139
1.33 Slide 33	140
1.34 Implicit Cursor Attributes	141
1.35 Implicit Cursor	142
1.36 Slide 36	143
1.37 Slide 37	144
1.38 Explicit Cursor Control	145
1.39 Example-1	146
1.40 Assume we have two tables- EMP(Empno, Ename, Sal, Deptno) Deptno references DEPT & DEPT(Deptno, Dname, Budget);	147
1.41 Slide 41	148
1.42 Slide 42	149
1.43 Slide 43	150
1.44 Explicit Cursor- cursor for loop	151
1.45 Parameterized Cursor	152
1.46 Example 3-Parameterized Cursor	153
1.47 Parameterized Cursor	154
1.48 Slide 48	155
1.49 END-PLSQL & CURSOR	156
<b>7. Exception_Function_Procedure_Package DSE 2023</b>	<b>157</b>
<b>1 24B7CoTF_hfsG3d_0</b>	<b>157</b>
1.1 EXCEPTIONS,PROCEDURES, FUNCTIONS, PACKAGES	157
1.2 Errors	158
1.3 Exception Handling	159
1.4 Slide 4	160
1.5 Predetermined Internal PL/SQL Exceptions(Built –in Exceptions)	161
1.6 Predetermined Internal PL/SQL Exceptions(Built –in Exceptions)	162
1.7 Slide 7	163
1.8 Slide 8	164
1.9 User Defined Exceptions	165
1.10 User Defined Exceptions	166
1.11 Slide 11	167
1.12 Slide 12	168
1.13 Slide 13	169
1.14 Slide 17	170
1.15 Slide 18	171
1.16 Slide 19	172
1.17 Using System Defined Numbered Exception	173
1.18 Using System Defined Numbered Exception	174
1.19 Slide 22	175
1.20 Slide 23	176
1.21 Slide 29	177
1.22 Procedures and Functions	178
1.23 Procedures and Functions	179

1.24	Slide 32	180
1.25	Procedures	181
1.26	CREATE PROCEDURE Syntax	182
1.27	Compiling Procedure	183
1.28	Showing Compilation Errors & Execution	184
1.29	Slide 37	185
1.30	Parameters	186
1.31	Defining the IN, OUT, and IN OUT Parameter Modes	187
1.32	Slide 40	188
1.33	Parameter Constraint Restrictions	189
1.34	Procedure with No Parameters	190
1.35	Executing DisplaySalary Procedure	191
1.36	Passing IN and OUT Parameters..	192
1.37	..Passing IN and OUT Parameters	193
1.38	Example 13.6 – Calling DisplaySalary2	194
1.39	Example– Using Bind Variables	195
1.40	Dropping a Procedure	196
1.41	Create Function Syntax	197
1.42	Example– No Parameters in Function	198
1.43	Example – Testing RetrieveSalary Function	199
1.44	Function with Parameter (Assume the table Employee(Empno, Firstname, MiddleName, LastName);	200
1.45	Function with Parameter	201
1.46	Testing FullName Function	202
1.47	Testing FullName Function	203
1.48	Dropping a Function	204
1.49	PACKAGE	205
1.50	PACKAGES	206
1.51	Package Specification and Scope	207
1.52	Create Package Specification Syntax	208
1.53	Declaring Procedures and Functions within a Package	209
1.54	Package Body	210
1.55	Create Package Body Syntax	211
1.56	Slide 64	212
1.57	Slide 65	213
1.58	Slide 66	214
1.59	Slide 67	215
1.60	Example Package	216
1.61	Package Body	217
1.62	Example– Package Body	218
1.63	Calling Package Procedure/Function	219
1.64	Results of Calling Package Procedure	220
1.65	Cursors in Packages	221
1.66	Cursors in Packages – Package body	222
1.67	Executing Cursors in Packages	223
1.68	END	224

<b>8. DATABASE TRIGGERS 2023</b>	225
1 38BECoTF_97Oi0P_0	225
1.1 DATABASE TRIGGERS	225
1.2 Database Trigger	226
1.3 USES-DATABASE TRIGGERS	227
1.4 Difference between Trigger and Constraints	228
1.5 Slide 5	229
1.6 Create Trigger Syntax	230
1.7 Trigger Types	231
1.8 Conditional Predicates for Detecting Triggering DML Statement	232
1.9 Slide 9	233
1.10 ROW Trigger – Accessing Rows	234
1.11 Bind Variables :old and :new Defined	235
1.12 Example	236
1.13 Example-2	237
1.14 WHEN clause- to specify condition under which Trigger has to fire	238
1.15 Example-3	239
1.16 STATEMENT Trigger - Example	240
1.17 STATEMENT Trigger - Example	241
1.18 Using Correlation Variable- Trigger - Example	242
1.19 Dropping a Trigger	243
1.20 A Cautionary Note	244
1.21 Temporarily enabling/disabling trigger	245
1.22 INSTEAD OF Trigger	246
1.23 INSTEAD OF Trigger	247
1.24 END	248
<b>9. Relational Database Design InComplete</b>	249
1 44C6CoTF_O1QOKR_0	249
1.1 Relational Database Design	250
1.2 Features of a Good relational database design	251
1.3 Update Anomalies	252
1.4 Example: Insert, Delete & Update Anomalies	253
1.5 Combine Schemas?	254
1.6 A Combined Schema Without Repetition	255
1.7 What About Smaller Schemas?	256
1.8 A Lossy Decomposition	257
1.9 Example of Lossless-Join Decomposition	258
1.10 Decomposition of a Relation	259
1.11 Goal — Devise a Theory for the Following	260
1.12 Decomposition using Functional Dependencies	261
1.13 Functional Dependencies (Cont.)	262
1.14 Example: Functional Dependency	263
1.15 Slide 16	264
1.16 Slide 17	265
1.17 Slide 18	266

1.18	Slide 19	267
1.19	Slide 20	268
1.20	Functional Dependencies Super Key & Candidate key	269
1.21	Example: Minimal Super Key	270
1.22	Example: Minimal Super Key (Contd.)	271
1.23	Example: Minimal Super Key (Contd.)	272
1.24	Write the Functional Dependencies for a given System Requirements	273
1.25	Slide 26	274
1.26	Set of Functional Dependencies that hold	275
1.27	Functional Dependencies(Cont'd)	276
1.28	Slide 29	277
1.29	Transitive dependency	278
1.30	Normalization	279
1.31	Normalization	280
1.32	Slide 33	281
1.33	Slide 34	282
1.34	First Normal Form	283
1.35	First Normal Form (Cont'd)	284
1.36	Example	285
1.37	1NF Decomposition	286
1.38	1NF Decomposition	287
1.39	Closure of a Set of Functional Dependencies	288
1.40	Closure of a Set of Functional Dependencies	289
1.41	Closure of Functional Dependencies (Cont.)	290
1.42	Slide 43	291
1.43	Procedure for Computing F+	292
1.44	Closure of Attribute Sets	293
1.45	Example for Attribute Set Closure	294
1.46	Slide 47	295
1.47	Slide 48	296
1.48	Exercises	297
1.49	Uses of Attribute Closure	298
1.50	1. Example Testing for superkey & Candidate Key:	299
1.51	Slide 52	300
1.52	Slide 53	301
1.53	Slide 54	302
1.54	2.Example for Testing functional dependencies	303
1.55	Exercises	304
1.56	3. Example for Computing F+	305
1.57	....Example for Computing F+	306
1.58	Slide 59	307
1.59	Exercises	308

# **DATABASE MANAGEMENT SYSTEM**

## **DSE**

## **IV Semester**

## **JAN 2023**

**Database System Concepts**

Abraham Silberschatz, Henry F. Korth, S. Sudarshan

# **Entity-Relationship Model**

# **ER-Model**

- Used for Conceptual Modeling of the Database schema.
- Meant for Human comprehension
- To tell End User, what is understood about their Domain.
- High-level database design without implementation details.
- DBMS independent.
- Building Blocks: **Entity, Attribute and Relationship**

# Entity

- An **entity** is an object that exists in the real world and is **distinguishable** from other objects.
- In real world, an entity has a set of properties and the **values for those properties identify** it.
  - Example: Faculty, Student, company, event, plant ,Course, Account
  - Properties of Student –RegNo, Name, Course, Phone.
  - Values of these property-(180370123,'Rajesh','MCA',9876999756) is an **entity**

# Entity Sets

- An **entity set** is a set of entities of the same type that share the same properties.
  - Example: set of all students who share same properties.
  - $\{(200970123, \text{'Rajesh'}, \text{'MCA'}, 9876999756), (200968124, \text{'Ramesh'}, \text{'DSE'}, 8876999756), \dots\}$
- Example:
  - Student entity set –Collection of all student entities.
  - Faculty entity set –Collection all faculties in a college

# Entity Sets- instructor and student

ID	name
76766	Crick
45565	Katz
10101	Srinivasan
98345	Kim
76543	Singh
22222	Einstein

*instructor*

ID	name
98988	Tanaka
12345	Shankar
00128	Zhang
76543	Brown
76653	Aoi
23121	Chavez
44553	Peltier

*student*

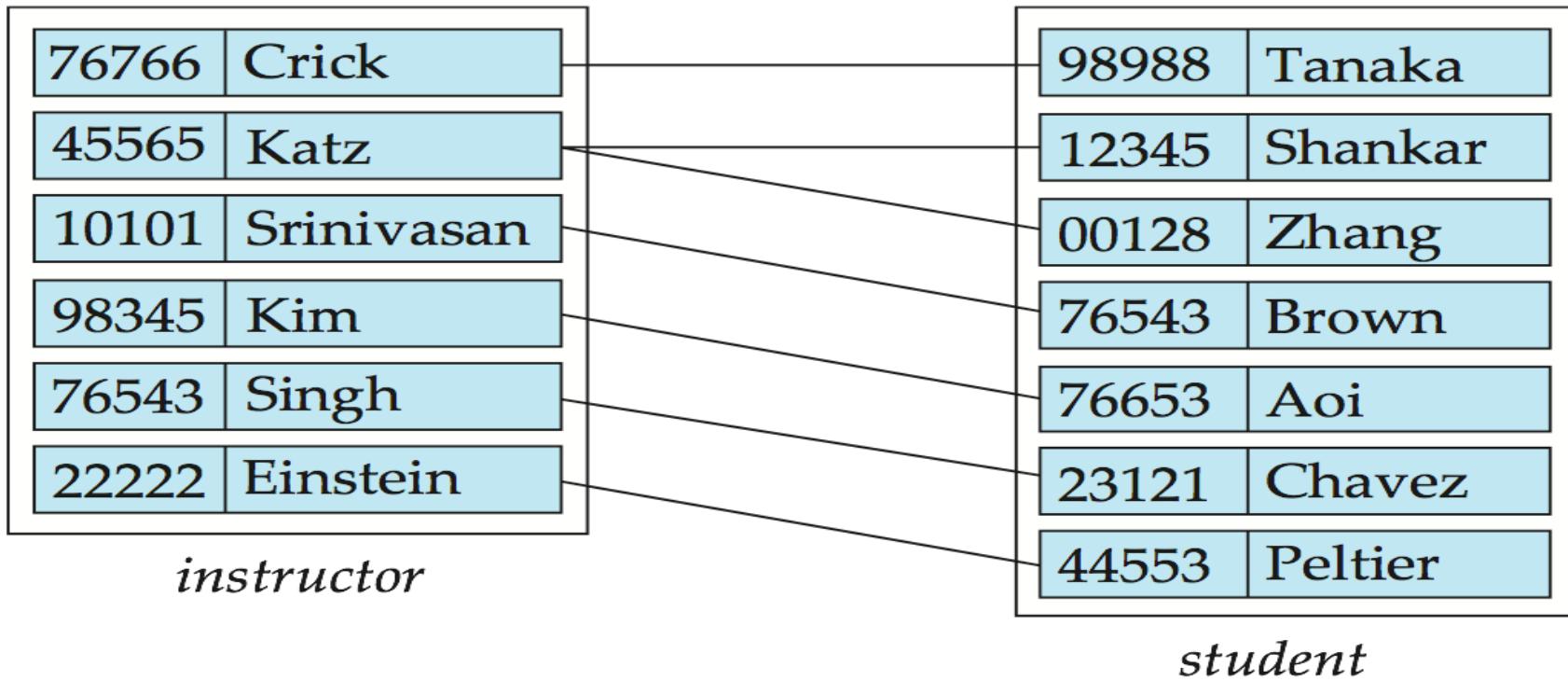
**Instructor=**{ (76766,Crick),45565,Katz),(10101,Srinivasan),...,(22222, Einstein) }

**Student=**{ (98988,Tanaka),(12345,Shankar),(00128,Zhang),....(44553,Peltier) }

# Relationship Sets

Assume that – we want to model the information such as – *Crick is advisor for Tanaka ; Katz is advisor for Shankar & Zhang ; Srinivasan is advisor for Brown*; and so on.

In other words, we are interested in representing information that **Some instructor will be advisor for Some Student/Students**(Association between Instructor & Student).



Relationship set = { (76766,Crick , 98988,Tanaka) , (45565,Katz,12345,Shankar),  
(45565,Katz,00128,Zhang),(10101,Srinivasan),... (22222, Einstein,44553,Peltier) }

# Relationship Sets

Consider formal definition of Relationship set.

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

Example:

(22222 ,Einstein) -I1

*instructor entity*

advisor

*relationship set*

(44553 ,Peltier)-S1

*student entity*

**(I1,S1)**

**Collection of such relationship is Relationship set**

- A **relationship set** is a mathematical relation among  $n \geq 2$  entities, each taken from entity sets  $E_1, E_2, \dots, E_n$ .

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

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

i.e. (44553 ,Peltier, 22222, Einstein) a relationship set element(instance)

*Every instance in a entity set is distinguishable from other entities using **primary key** , in the same way every relationship in a relationship set is distinguishable.*

**Primary Key of relationship set is –  $PK(E_1) \cup PK(E_2) \cup \dots \cup PK(E_n)$**

Example:

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

# Relationship Sets (Cont.)

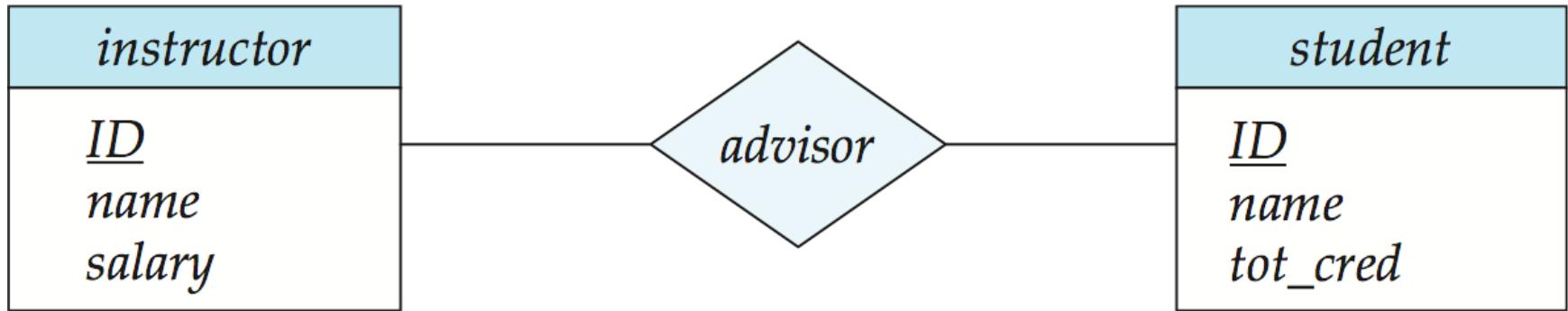
- The association between entity sets is referred to as **participation**.
- A **relationship instance** (ex: (45565,Katz-00128,Zhang)) in an E-R schema represents an **association between the named entities** (ex: (45565,Katz) an Instructor entity & (00128,Zhang) a Student entity) in the real-world enterprise that is being modeled.
- The function that an entity plays in a relationship is called entity's **role**.

**Relationship set Advisor** ={ (76766,Crick - 98988,Tanaka) , (45565,Katz- 12345,Shankar),  
(45565,Katz-00128,Zhang),...,... (22222, Einstein-44553,Peltier) }

To identify every relationship instance in the relationship set distinctly Primary Key of relationship set is used.  **$PK(\text{Advisor}) = PK(\text{Instructor}) \cup PK(\text{Student})$**

# E-R Diagrams

Following is the diagrammatic representation of Relationship( discussed in slide 11) in ER Model



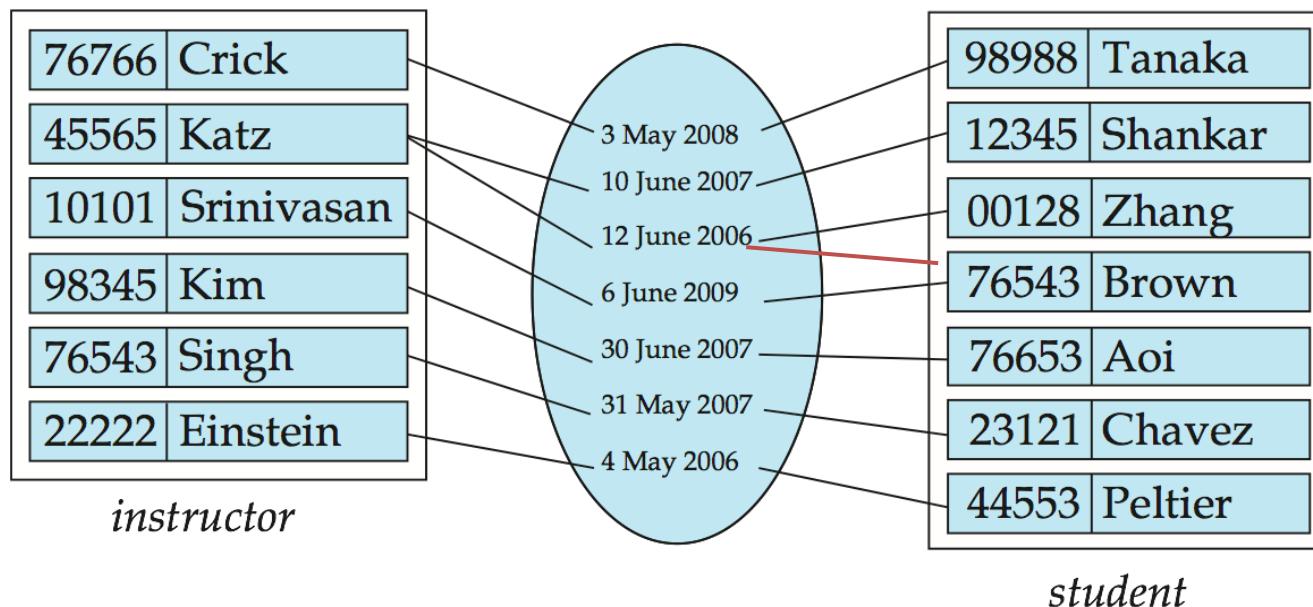
- Rectangles represent **entity sets**.
- Diamonds represent **relationship set**.
- **Attributes** listed inside entity rectangle.
- Underline indicates **primary key** attributes.

# Relationship Sets (Cont.)

- A relationship may also have attributes called **descriptive attributes**.

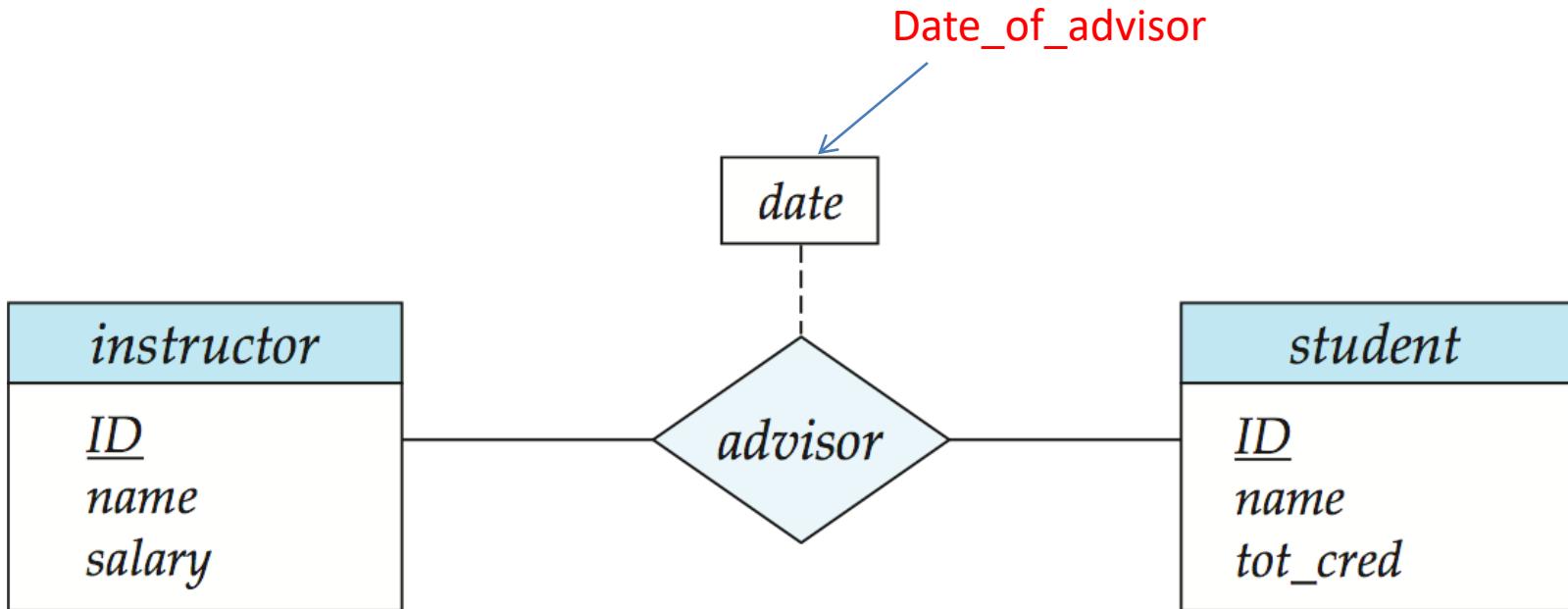
Assume that we want to store information that –An Instructor **I** became an Advisor to Student **S** on the date **D**.

**Ex:** (76766,Crick) became Advisor to a Student (98988,Tanaka) on 3<sup>rd</sup> May 2008.



3<sup>rd</sup> May 2008 date is property of neither Instructor nor Student, but it has meaning when referred w.r.t Advisor relation ship between Instructor & Student.

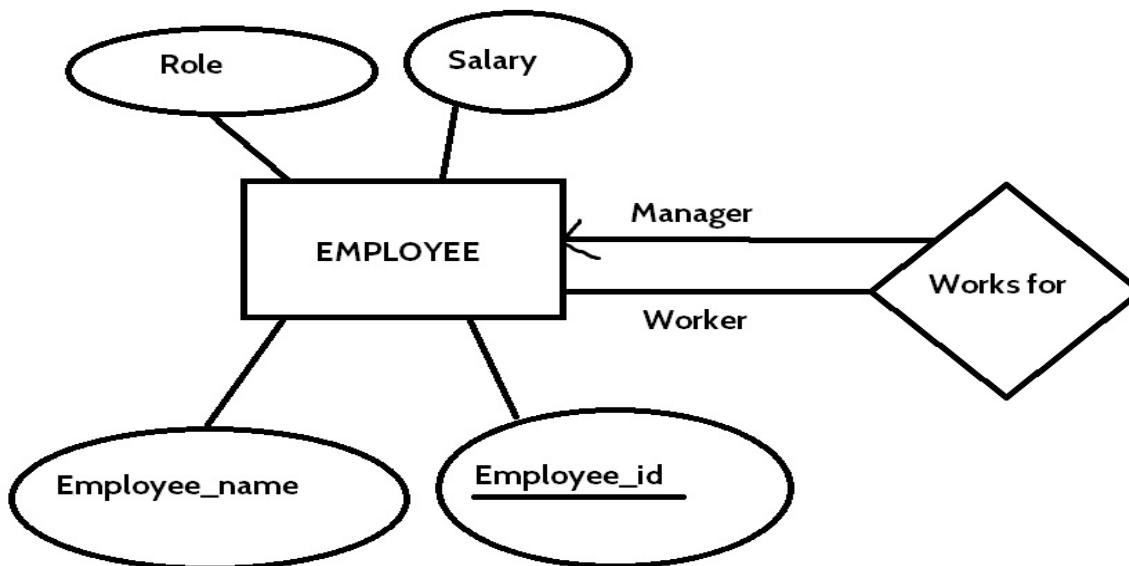
# Relationship Sets with Attributes



The design decision of where to place descriptive attributes in such cases—as a relationship or entity attribute—should reflect the characteristics of the enterprise being modeled.

# Recursive Relationship set

- If the same entity sets participates in a relationship set more than once in different roles- Recursive relationship



# Recursive Relationship..

- Example: Sample relationship between EMP entities.

Emp_ID	Ename	Job	Salary
100		Clerk	
101		Manager	
103		S.Clerk	
108		Accountant	
105		O.Assitant	
108		S.Manager	
109		R.Manger	

The diagram illustrates a recursive relationship within the EMP entity set. Blue arrows originate from the Emp\_ID values 101, 108, and 109, and point back to the same row (the second, fifth, and sixth rows respectively) in the Emp\_ID column. This visualizes how an entity can have a relationship with itself, which is the core concept of a recursive relationship.

(100,..,Clerk,..) is an entity in EMP entity set and is having relationship (association) with another entity (101,..,Manger,..) in the same Entity set( i.e. EMP)

**Recursive Relationship** when converted into schema, it looks as below-

**ManagerNo** is foreign key Referencing EMP which models the recursive relationship shown in ER diagram.

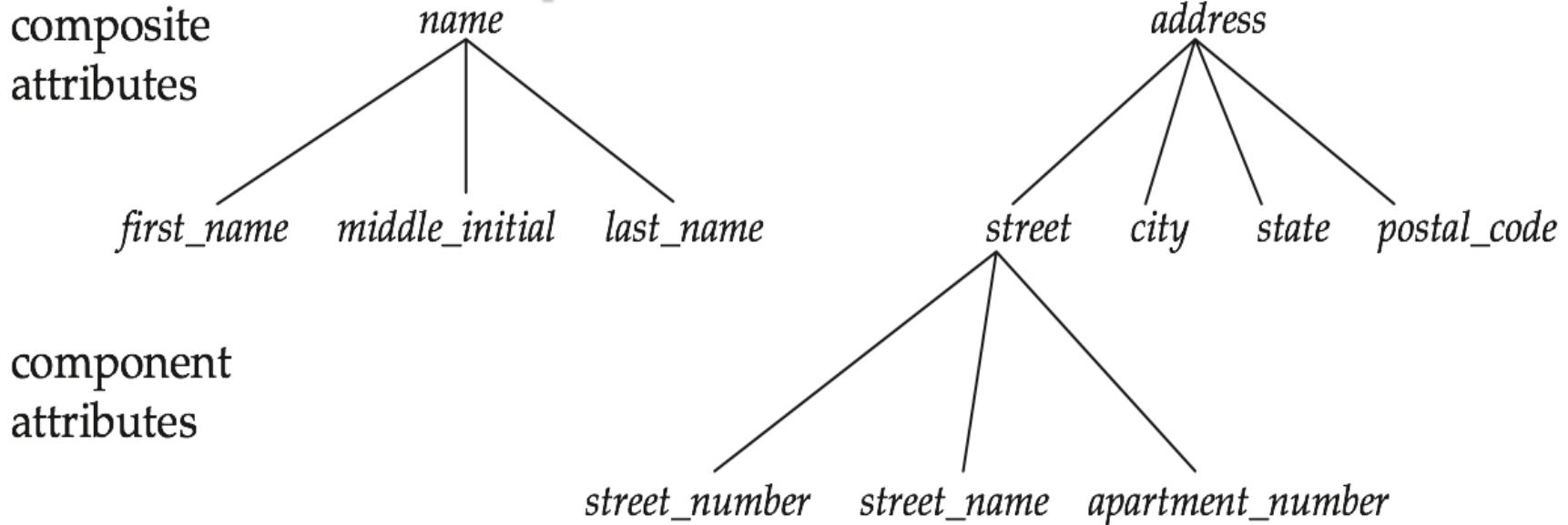
We will study, how ER diagram is converted into schema(set of Tables) later in this chapter.

Emp_ID	Ename	Job	Salry	ManagerNo
100		Clerk		101
101		Manager		
103		S.Clerk		101
108		Accountant		109
105		O.Assitant		
108		S.Manager		
109		R.Manger		109

# 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.
    - **Simple-** having atomic or indivisible values.
      - Example: Department\_Name , State, city
    - **Composite-** having several components in the value.
      - Example: Contact\_number attribute further having components-  
STDCode and Phone Number

# Composite Attributes



## Single-valued and multivalued attributes

**Single**- having only one value rather than a set of values.

Example: *Birth Place*

**Multivalued** - having a set of values rather than a single value

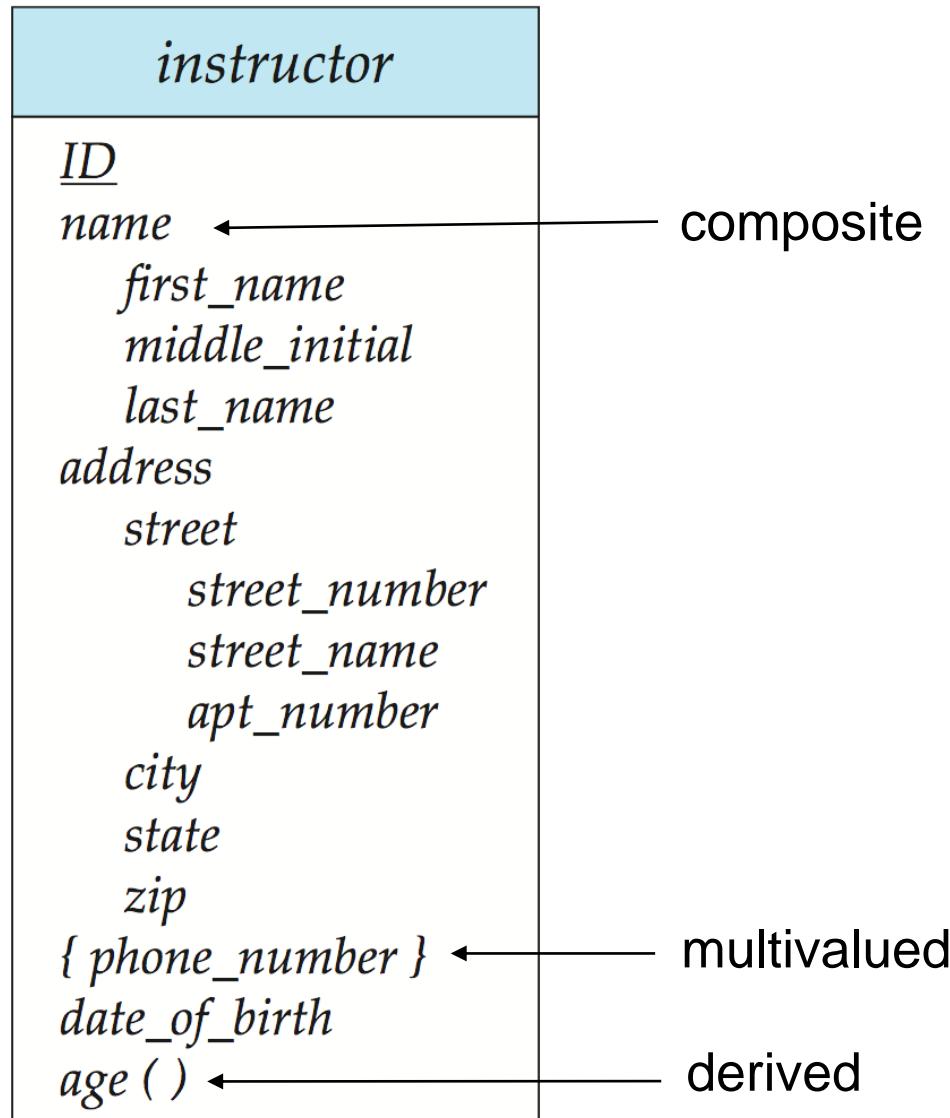
Example: *phone\_numbers ,Emails of a person , Degrees acquired by a faculty*

## Derived attributes

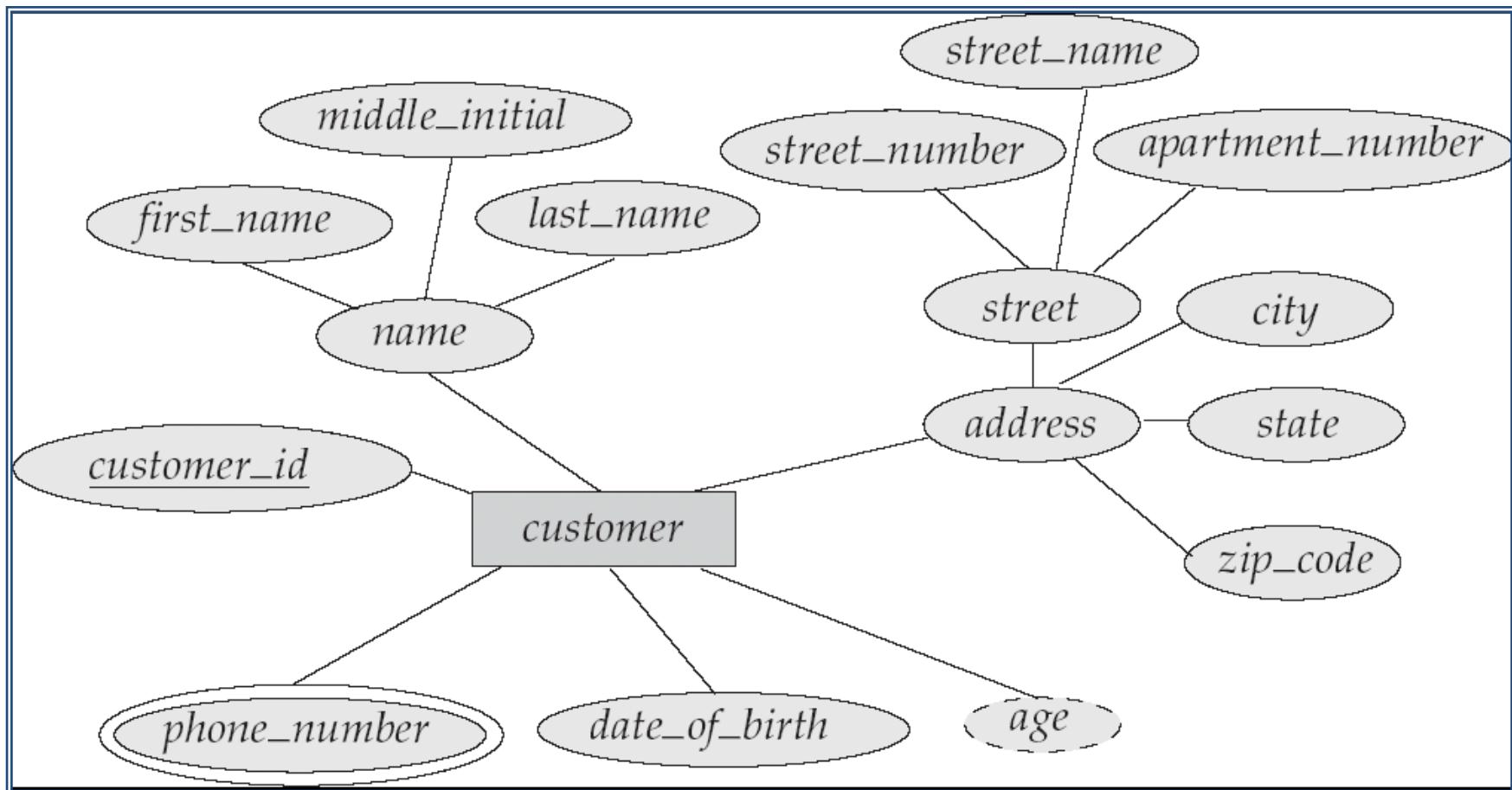
Can be computed from other attributes

Example: *Age, given date\_of\_birth*

# Entity With Composite, Multivalued, and Derived Attributes

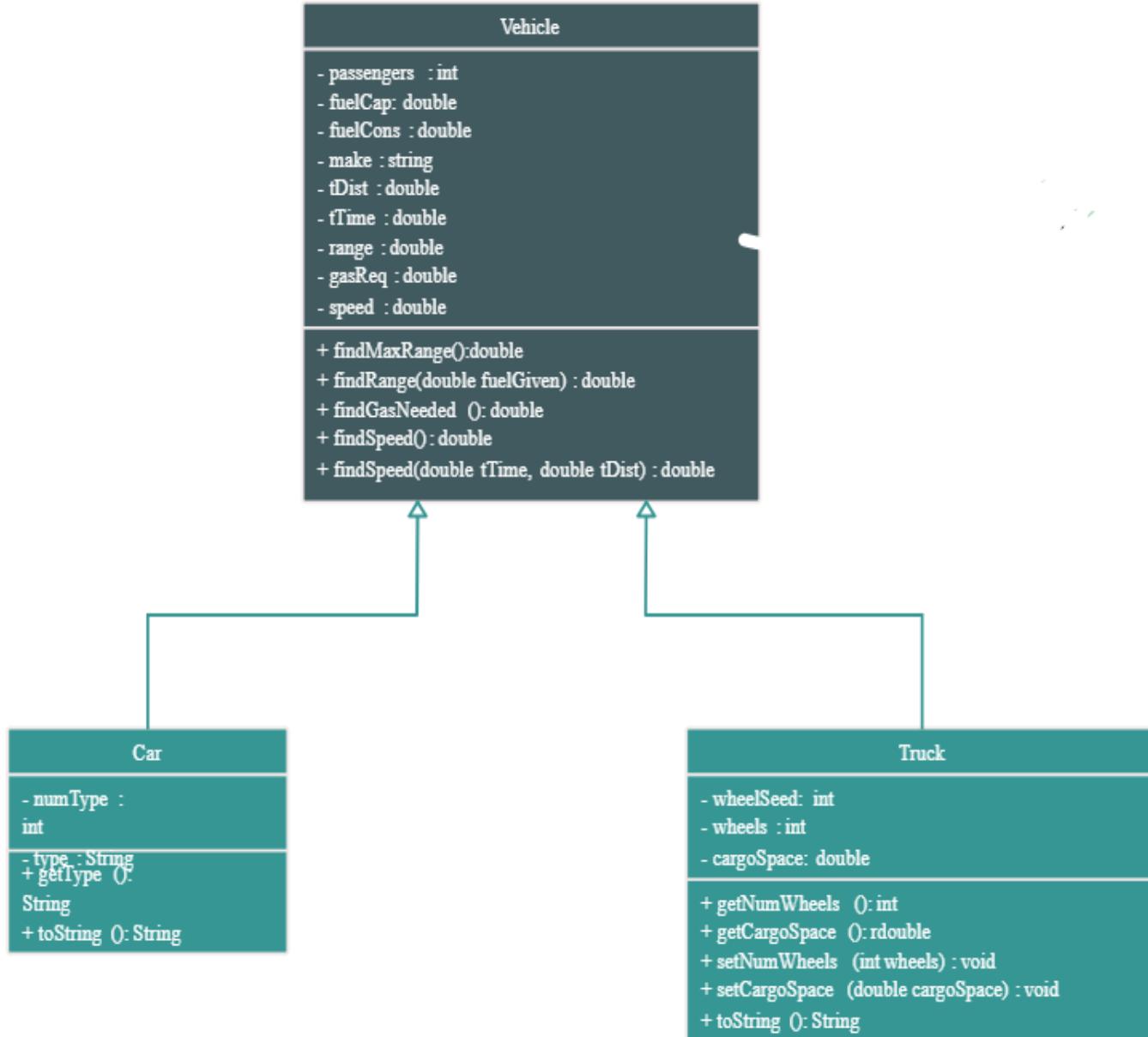


# Entity With Composite, Multivalued, and Derived Attributes



In 4<sup>th</sup> Edition

# Similar to UML- Class Diagram



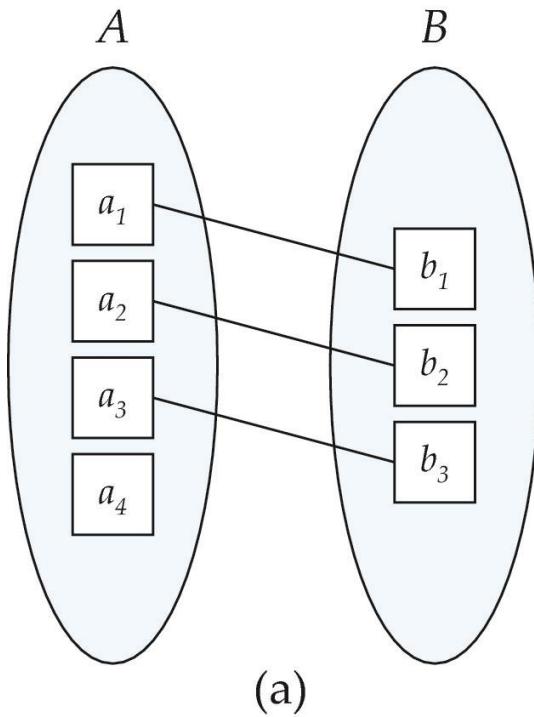
# Degree of a relationship

- **Degree:** the number of participating entity sets in a relationship.
  - Degree 2: *binary*
    - involve two entity sets
  - Degree 3: *ternary*
  - Degree n: *n-ary*
- Binary relationships are very common and widely used.

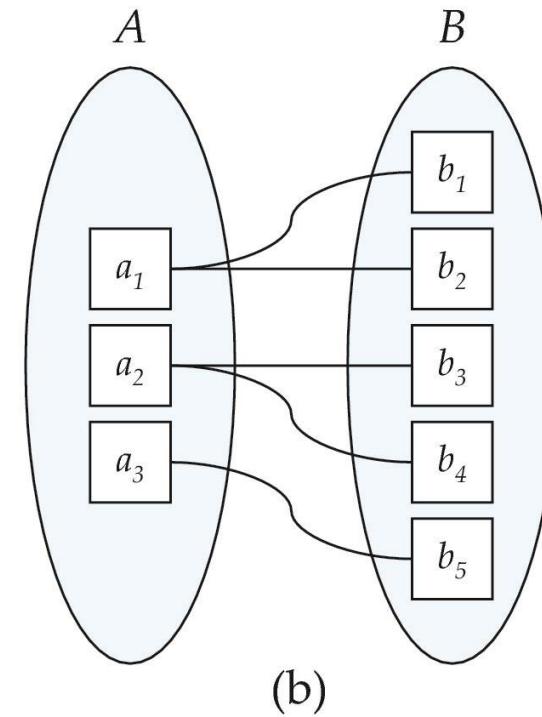
# Mapping Cardinality Constraints

- 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



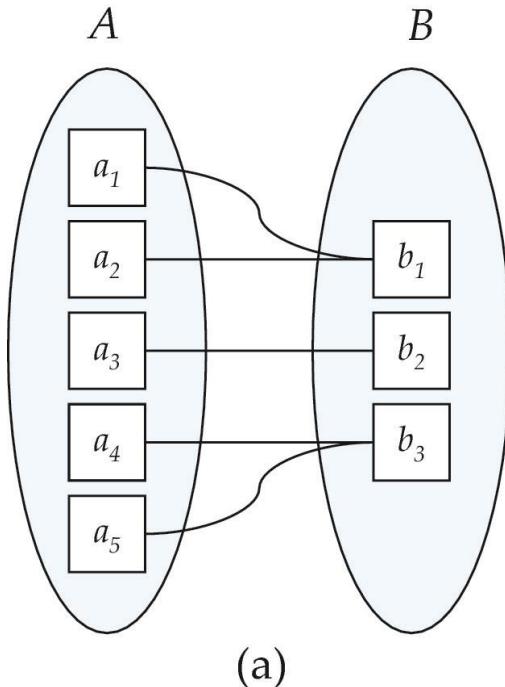
One to one



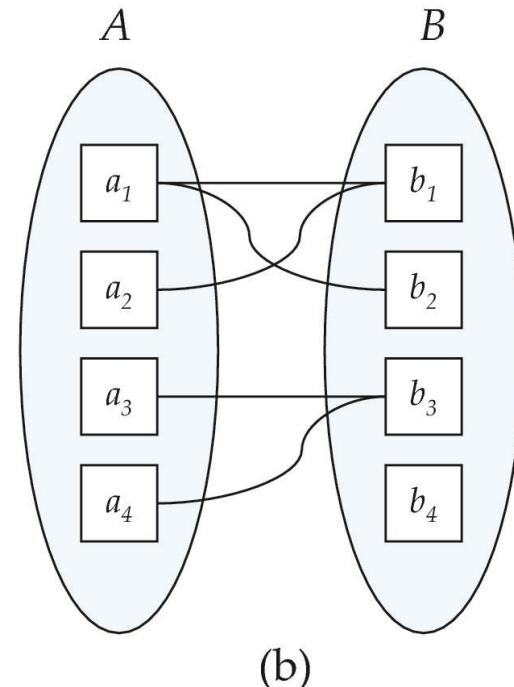
One to many

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

# Mapping Cardinalities



Many to one



Many to many

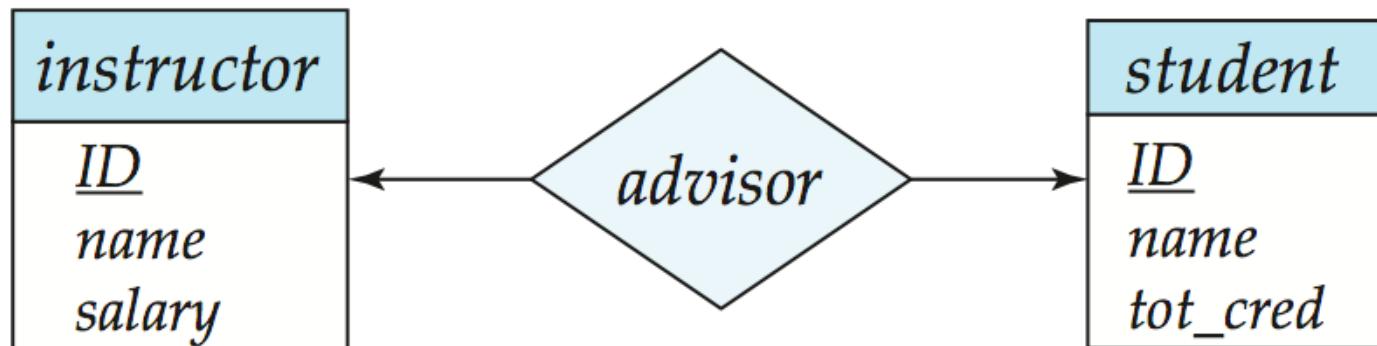
Note: Some elements in A and B may not be mapped to any elements in the other set

# 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.

# 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*”



# Example one-one

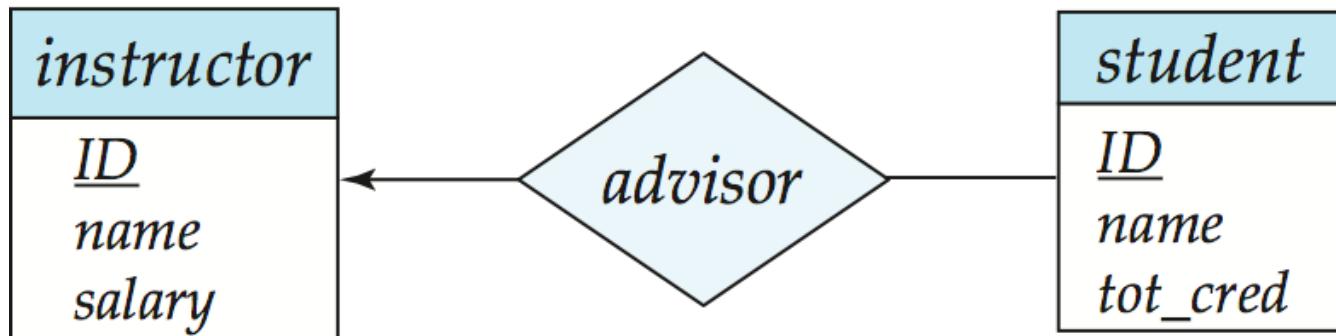
Institute		Head_Of_Institute		
Institute Name	Location	Head_Name	Experience	Qualification
ABC Tech	Mangalore	Rajesh	28	Mtech PhD
AAA Engg College	Bangalore	Ravi	30	MSc PhD
BBB Technology	Hyderabad	Blake	19	MS PhD
VTU	Belgaum	Mukhesk	25	Mtech PhD

A sample **one-to-one** relationship between an *Institute* and a *Head\_Of\_Institute*

**1 Institute will have only 1 Head of Institute –Therefore exactly one Head\_Of\_Institute entity is associated with exactly one Institute Entity**

# 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* relationship.
  - a *student* is associated with **at most one** *instructor* via *advisor*,



# Example one- Many

**Customers**

CustID	Name	City	PAN_NO
C101	Rajesh	Manipal	ABC1028
C102	Ravi	Mangalore	AFC1097
C103	Blake	Bangalore	BSD2013
C104	Mukhesk	Udupi	RES2001
C105	Ram	Udupi	FGU7629

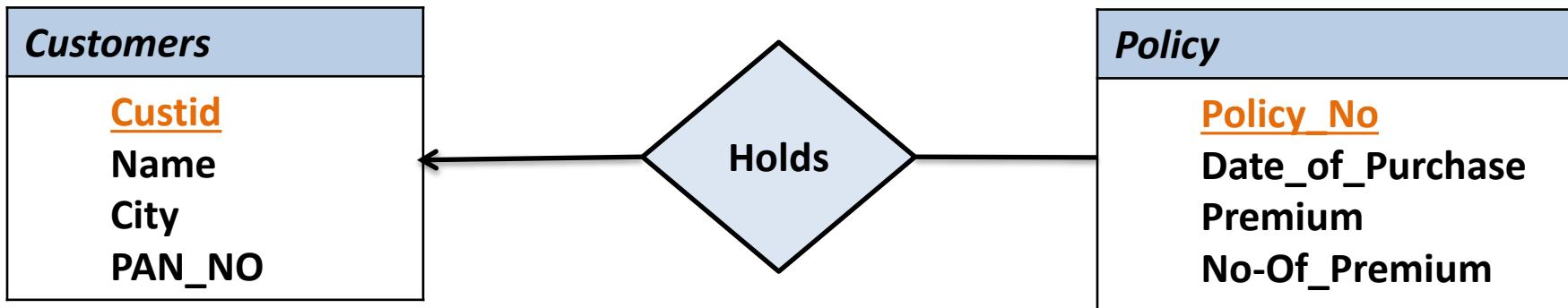
**Policy**

Policy_No	Date_of_Purchase	Premium	No_of_Premium
LIC101	10-10-2001	4000	40
LIC102	1-1-1990	5000	30
LIC107	2-5-2012	7000	20
LIC102	2-9-2012	7000	25
LIC105	10-4-2001	8000	40
LIC202	2-3-2003	9000	30
LIC321	2-5-1998	3000	25
LIC312	4-4-2001	4000	28
LIC383	9-1-2004	5000	15

a sample **one-to-many** relationship between

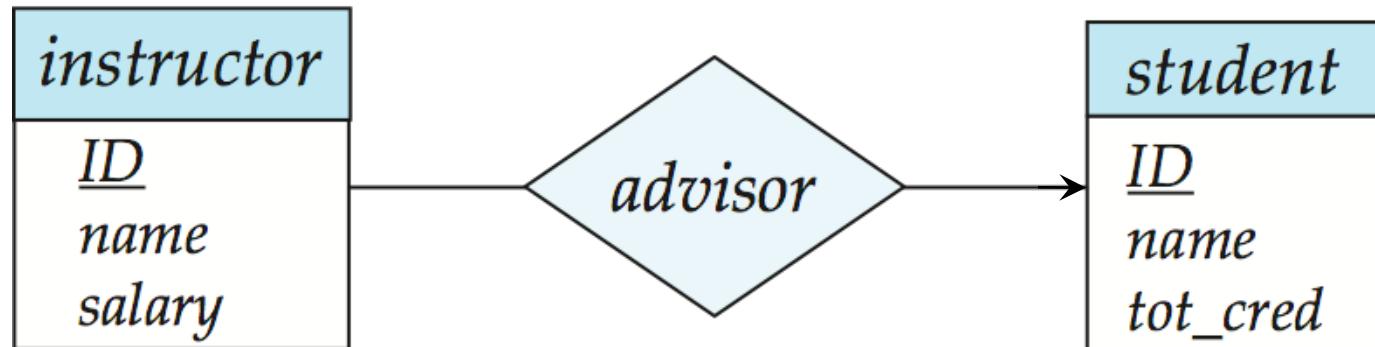
an **Customers** and a **Policy**

**1 Customer may take 1 or More LIC policies**



# 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*



## *Employee*

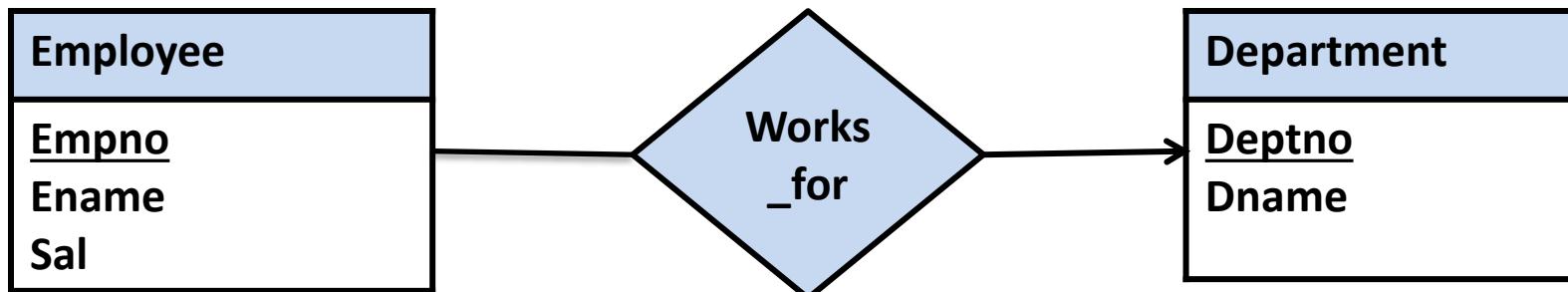
Empno	Ename	Sal
100	Ravi	2000
101	Raj	3000
102	Vikram	5000
103	Santhosh	4000
104	Ajay	7000
105	Anoop	8000
106	Sanoop	9000
107	Rakesh	7000

## *Department*

Deptno	Dname
D1	MCA
D2	CS
D3	IT

A sample **many-to-one** relationship between an *Employee* and a *Department*

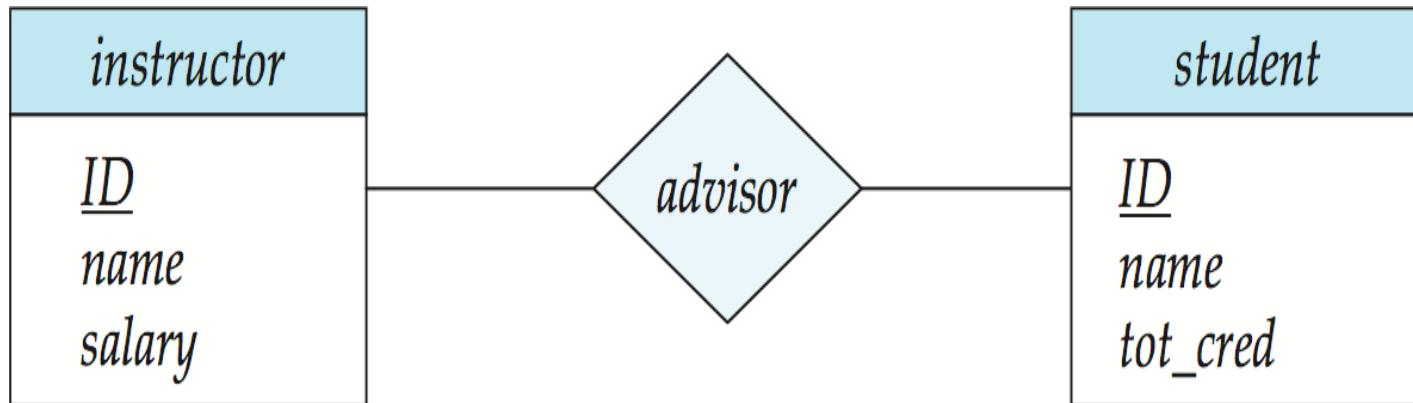
***More than 1 employees work in 1 Department***



One Department entity may be associated with one or more Employee entities

# 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*



# Example Many - Many

**STUDENTS**

RegNo	Name	Phone
MCA101	Rajesh	124478
MCA102	Ravi	344535
MCA103	Blake	473456
MCA104	Mukhesh	445987
MCA105	Ram	896949
MCA106	Vijay	800543

**SUBJECTS**

SubjectID	SubjectName	Credits
S101	OS	4
S102	OT	3
S103	INS	3
S104	ADBMS	4
S105	OOPD	4
S106	DS	4

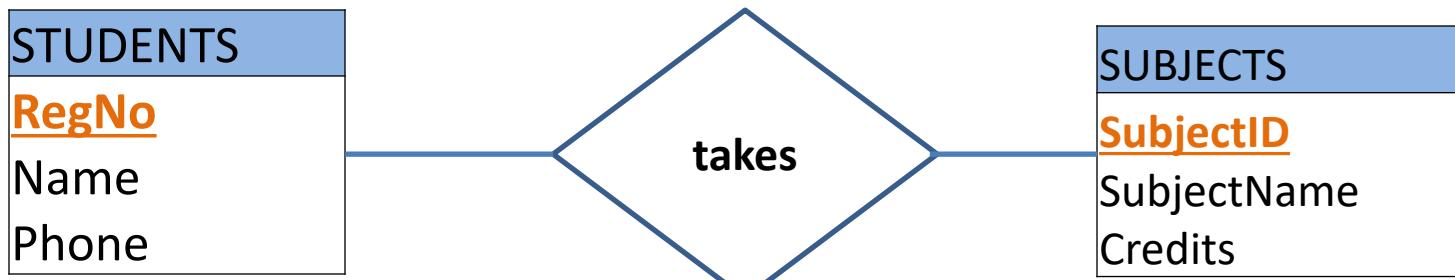
a many-to-many relationship between an **STUDENTS** and a **SUBJECTS**

**More than 1 Student can take 1 subject – therefore Many-1**

**More than 1 Subject can be taken by 1 student – therefore 1-Many**

**Equivalently**

Multiple students can take Multiple Subjects. Therefore **Many-Many**



## Example- Design ER Model

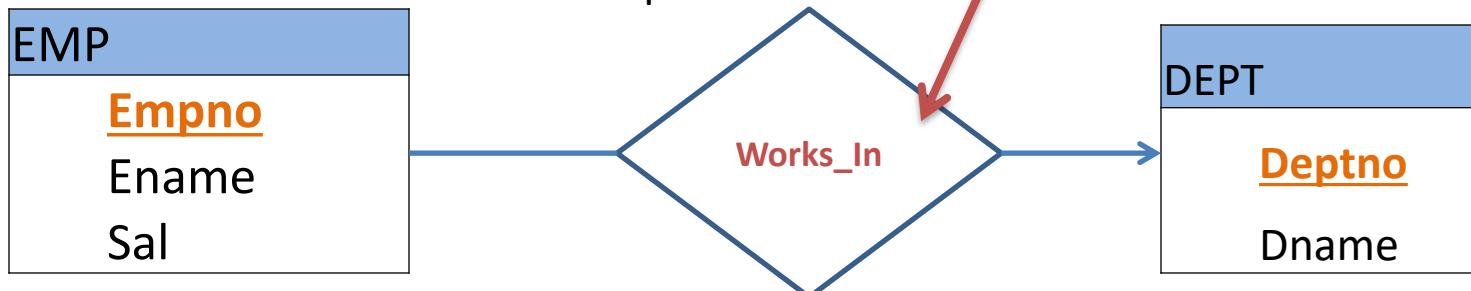
- Consider the following data requirement of an Institute. Assume that we want model the data requirements using ER modeling.
- The Institute is comprised of several departments such as – DSCA, CS, IT etc.. and each department has a Department Number such as D1,D2,.. used to identify each department. Each department has unique name. In every department many employees work. Each employee is identified by a unique Employee Number. Other information about each employee we need is employee name and salary.

Empno	Ename	Sal
100	Ravi	2000
101	Raj	3000
102	Vikram	5000
103	Santhosh	4000
104	Ajay	7000
105	Anoop	8000
106	Sanoop	9000
107	Rakesh	7000

## Entity Sets ,Sample Entities and relationship

Deptno	Dname
D1	MCA
D2	CS
D3	IT

A sample Employee and Department entities association is the Relationship set.  
and it is modelled as **Works\_In**  
Relationship below



The relationship is having Many-1 Cardinality . An employee can work at most in only 1 department.

**Note** that If you are adding **Deptno** as another column in EMP to represent Employee working in a department association, it becomes **redundant information** as **Works\_In** implicitly represents it. (Same is discussed under heading –[Redundant Attributes](#))

## Example- Design ER Model...

Assume that we want model following data requirements of Institute using ER modeling.

The college is comprised of several departments such as –DSCA,CS,IT etc.. and each department has a Department Number such as D1,D2,.. used to identify each department. In every department many employees work. Each employee is identified by a unique Employee Number. Information about each employee we need is employee name and salary.

Extend previous ER diagram with following Data Requirements.  
Assume appropriate attributes.

We also want to record information related to the courses offered by each department such as courses names, Number of semesters, Total Credits etc.

For example– IT department offers – Inform. & Tech., Computer & Comm. , DSCA offers DSE , MCA and Computer Science offers Inform. Security , MSc in CS.

## Entity Sets ,Sample Entities and relationship

Department Entity Set

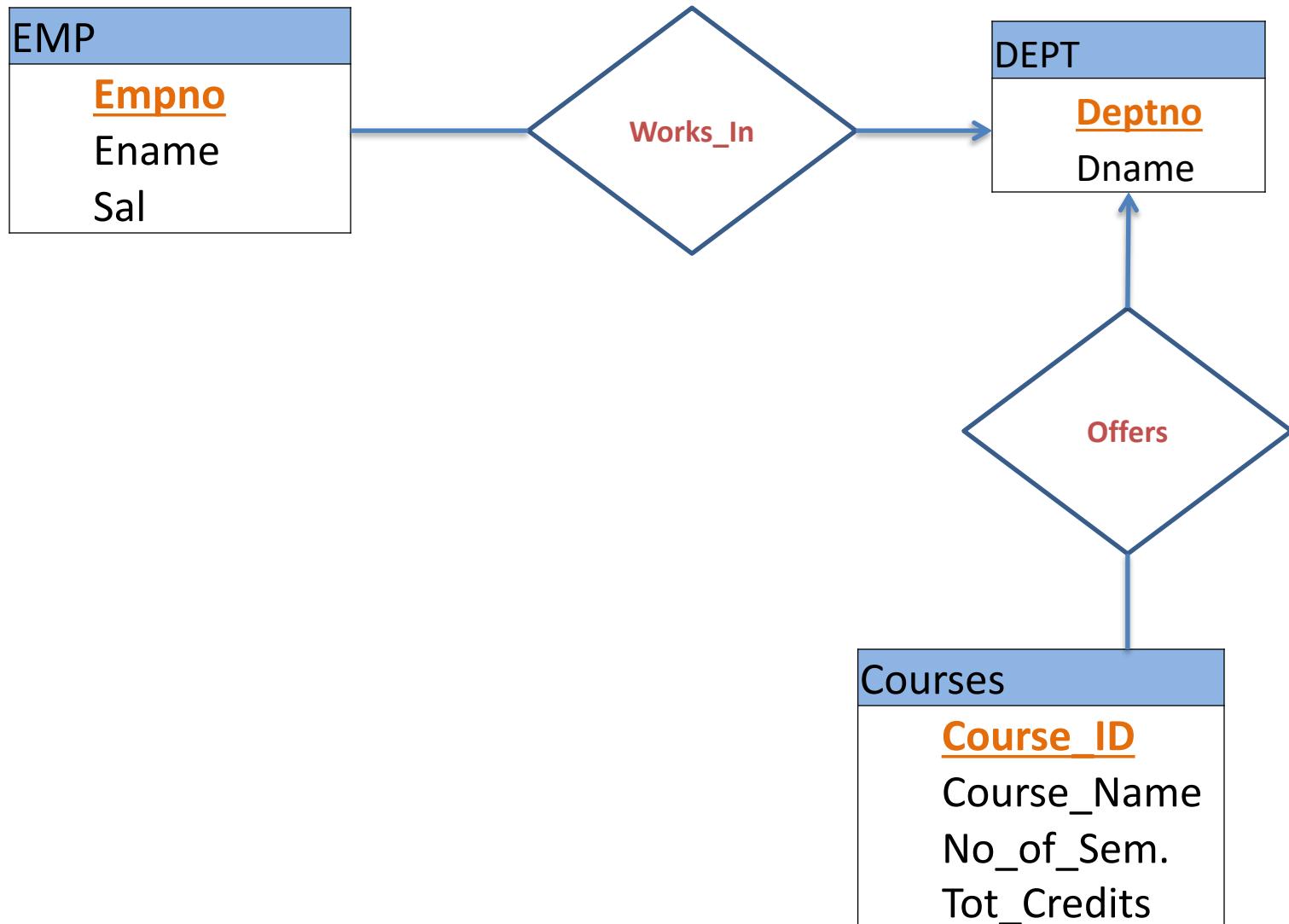
Deptno	Dname
D1	DSCA
D2	CS
D3	IT

Course Entity Set

Course_ID	Course_name	No-Of Sem	Credits
100	Info & Tech		
101	Comp&Com		
102	Info.Security		
103	Comp Science		
104	DSE		
106	MCA		

A sample association  
between Department  
entities and Course  
entities

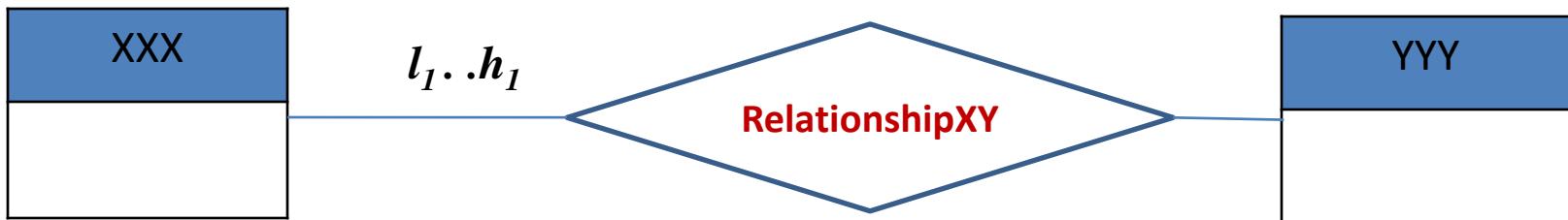
# ER Diagram for the requirements given



# Notation for Expressing More Complex Constraints

- Along with Cardinality constraints such as **1-1**, **1-M** & **M-M**, **Cardinality limits** can also be used express participation constraints.
- Cardinality limits tells about-
  - The number of times (Minimum & Maximum) each entity participates in relationships in a relationship set.
  - 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 **XXX** entity may be associated with a minimum of  $l_1$  number of **YYY** entities or at the maximum  $h_1$  number of **YYY** entities through **RelationshipXY** relationship.



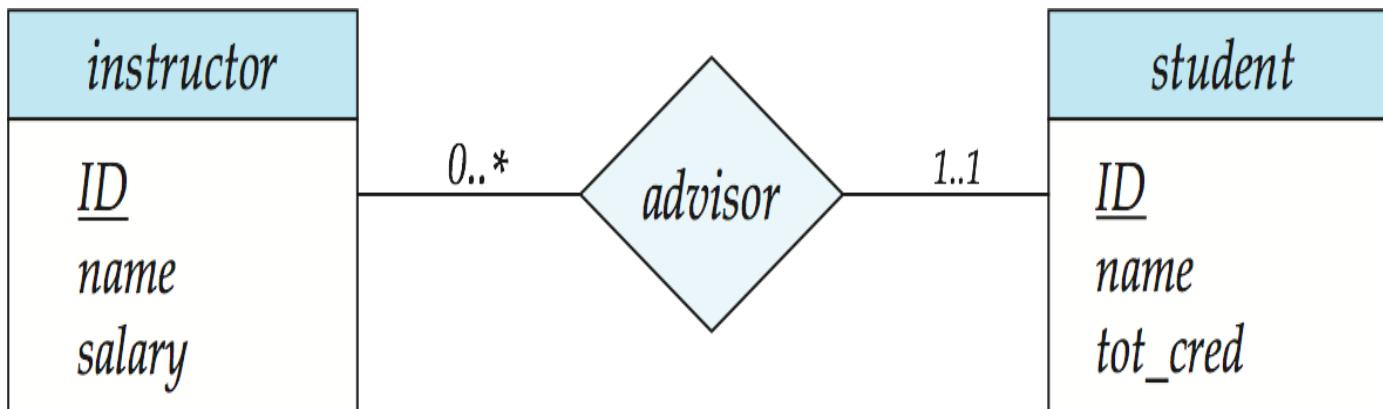
# Notation for Expressing More Complex Constraints

## Example:

- The line between advisor and student has a cardinality constraint of **1..1**, meaning the **minimum** and the **maximum** cardinality are both **1**.

A student entity is associated with max & min 1 instructor entity only.  
Means each student must have exactly one advisor.

- The limit **0..\*** on the line between *Instructor* and *Advisor* indicates that an instructor may be associated with **zero or more students**.  
Means instructor can be advisor for **zero or more** students.



## Instructor

ID	Name	Salary
I1		
I2		
I3		
I4		

## Student

ID	Name	Total_Credit
S1		
S2		
S3		
S4		
S5		

An Instructor **may or may not** be an Advisor.

Hence **Minimum** number of Instructor entity that can be associated with student entities through advisor is **0**.

An Instructor may be Advisor to any number of Students.

Hence **Maximum** number of Instructor entity that can be associated with student entities through advisor is **\***.

**0..\***

**Minimum-0** means every Instructor need not participate in Advisor.

i.e. **Partial participation**

**Advisor**

Each student must have **exactly one advisor**.

Every student entity is Associated with an Instructor via Advisor Relationship.

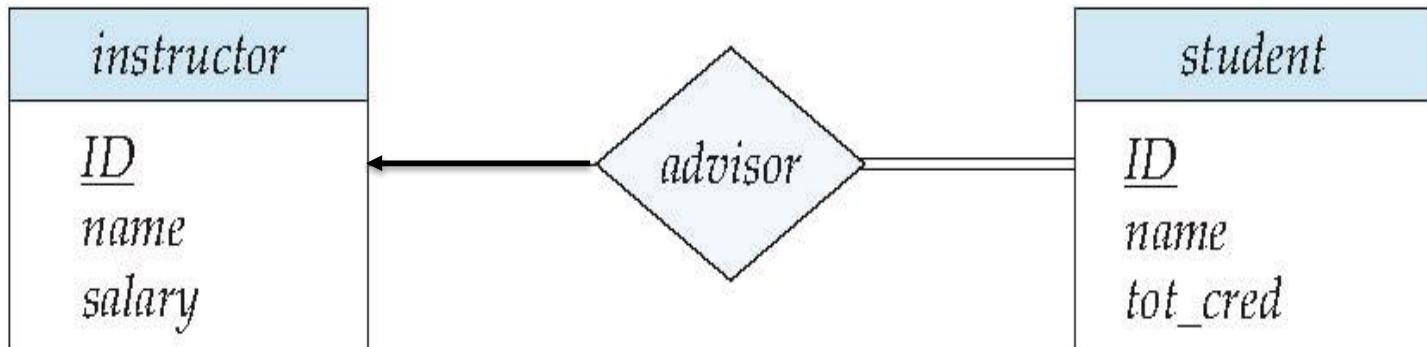
**Maximum=1 & Minimum=1** Student entity that can participate in Advisor is **1..1**

**Minimum-1** means every Student must participate in Advisor.

**Total participation**

# Participation of an Entity Set in a Relationship Set

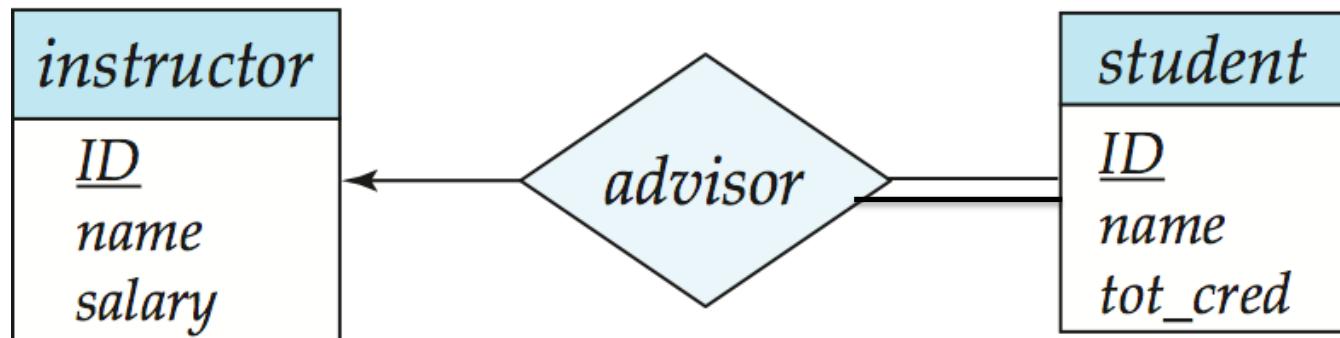
- **Total participation** (indicated by **double line**):  
every entity in the entity **set** participates in at least one **relationship** in the relationship set
- E.g., participation of *Student* in *advisor* is **total**
  - ▶ every *Student* must be associated with *instructor*

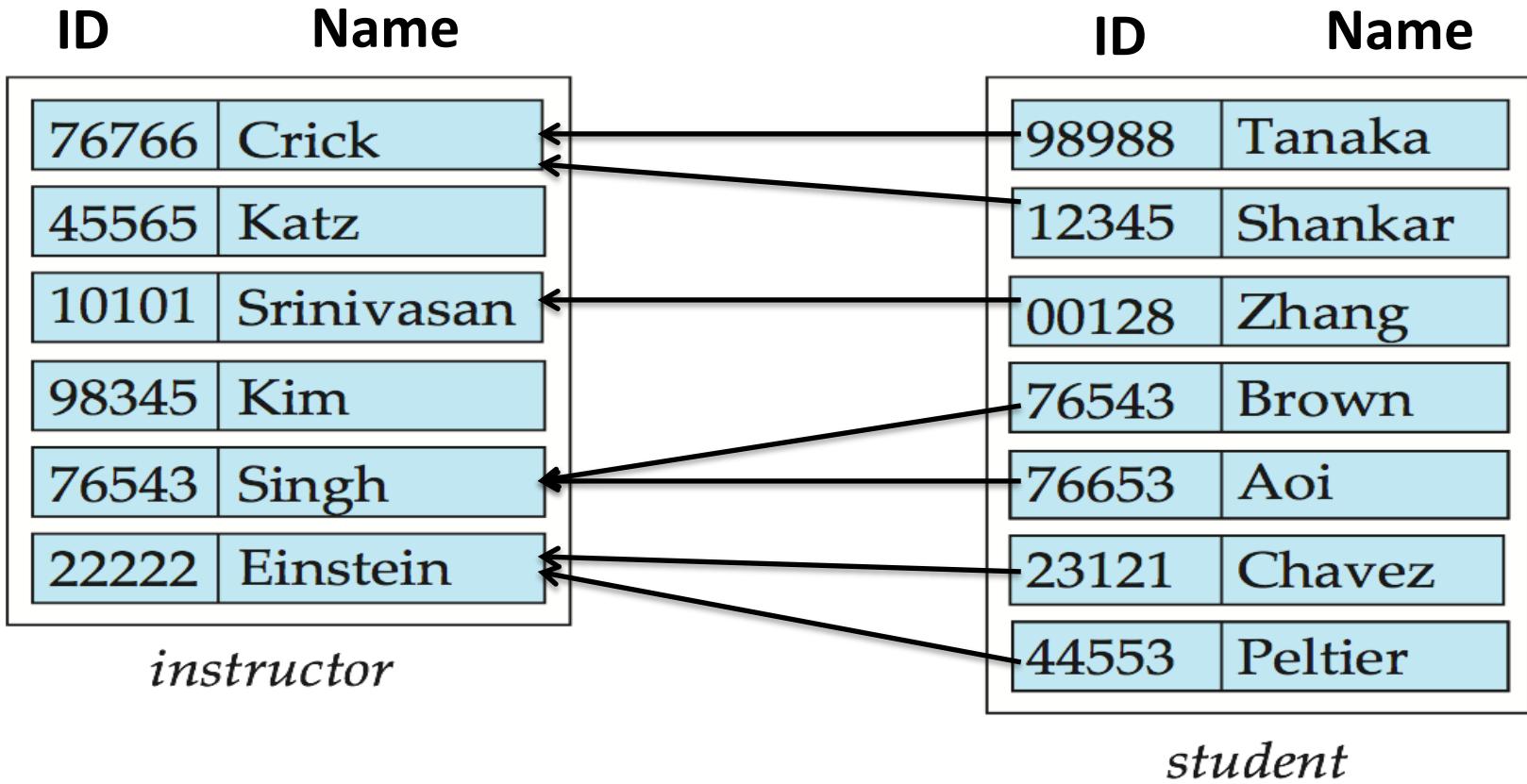


## □ Partial participation:

some entities **may not participate in any relationship** in the relationship set

- Example: participation of ***instructor*** in ***advisor* relationship** is **partial** (see next slide)





For an **Instructor**, being an Advisor is optional –

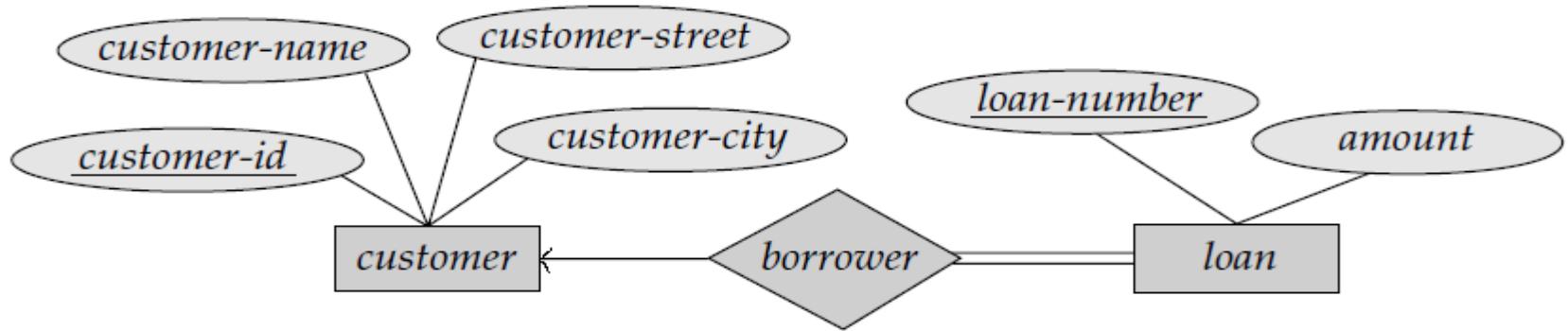
Therefore **Instructor** entity participation is **PARTIAL**)

**Note:** (45565,Katz) and (98345, Kim) are **not participating** in advisor relationship

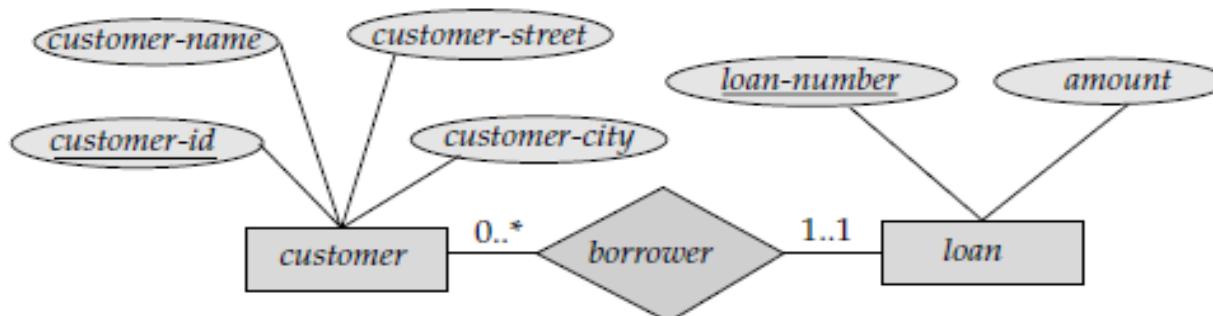
but Every student must have an Advisor –

Therefore **Student** entity participation is **TOTAL**.

# Total and Partial participation ...

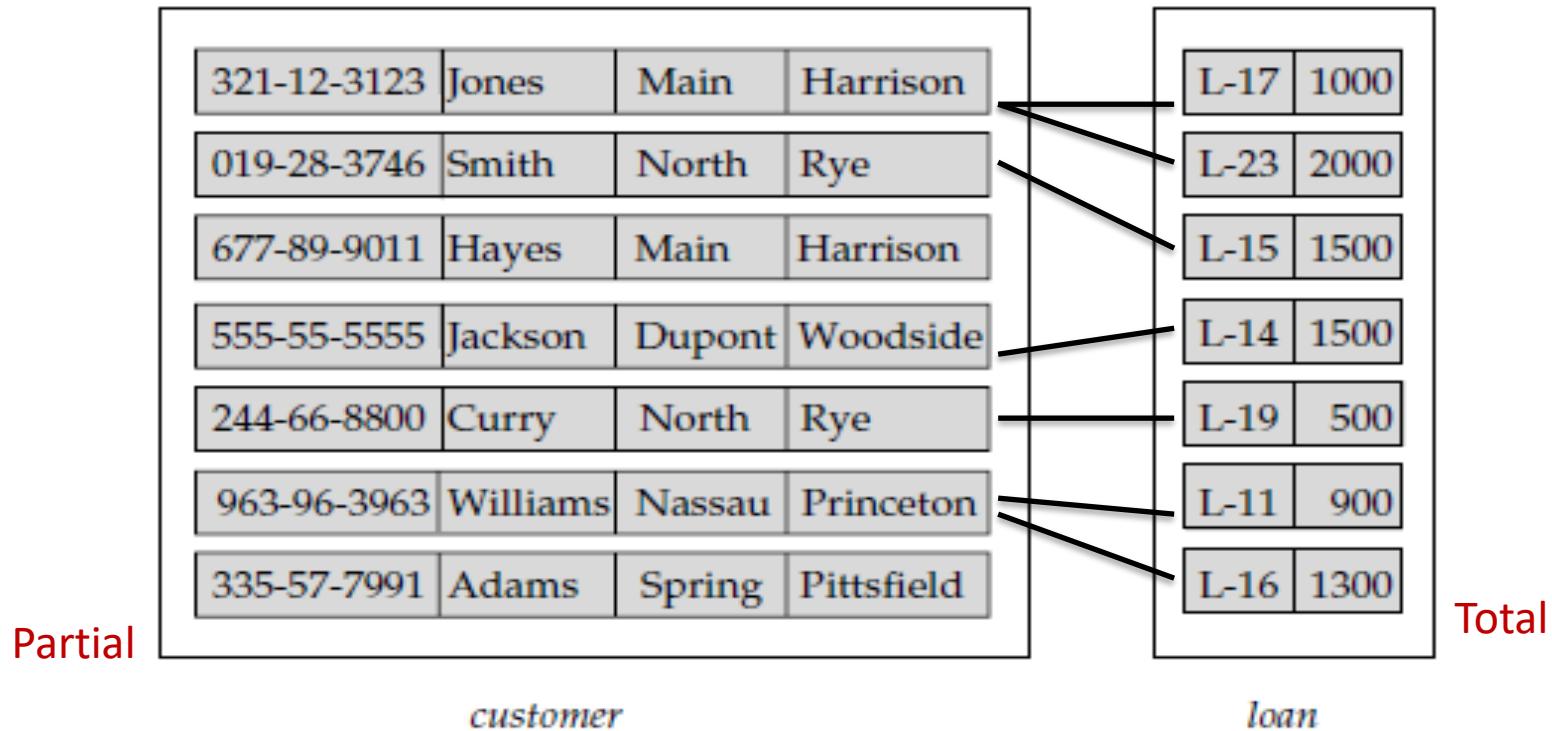


- participation of *loan* in *borrower* is **total**
  - In loan (entity type), every loan entity has to be associated with 1 or more customers, because a loan can't be given if there do not exist customer taking loan.
- participation of *customer* in *borrower* is **Partial**
  - In Customer, there may be some customers not taking loan and so do not participate in borrower relationship



Same ER diagram with  
Cardinality Limit for  
Participation

## Total and Partial participation ...



Partial

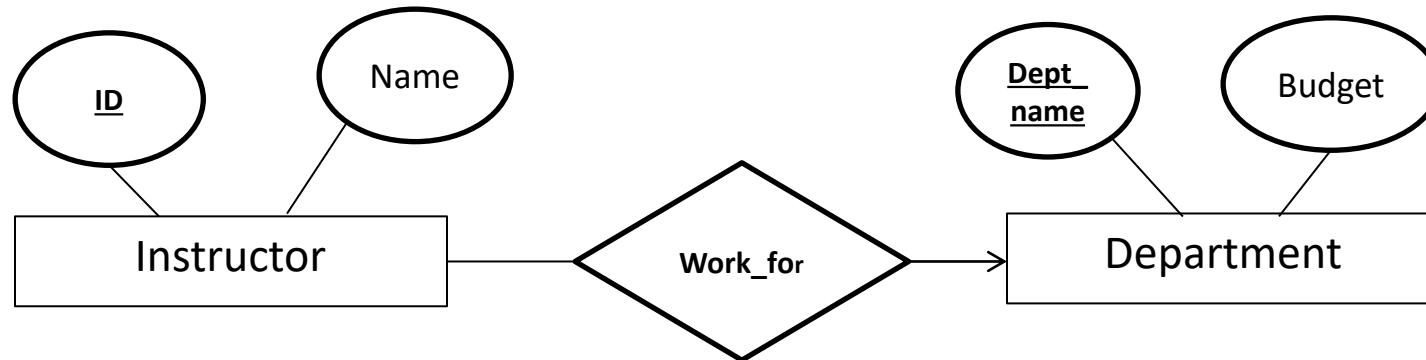
Total

Note that Customers- Hayes & Curry haven't taken any loan , therefore Customer participation is Partial in Borrower relationship.

But, a Loan has to be given to a customer only , so a Loan entity can not exist without being associated with Customer . So Loan participation is Total in Borrower relationship

# Redundant Attributes

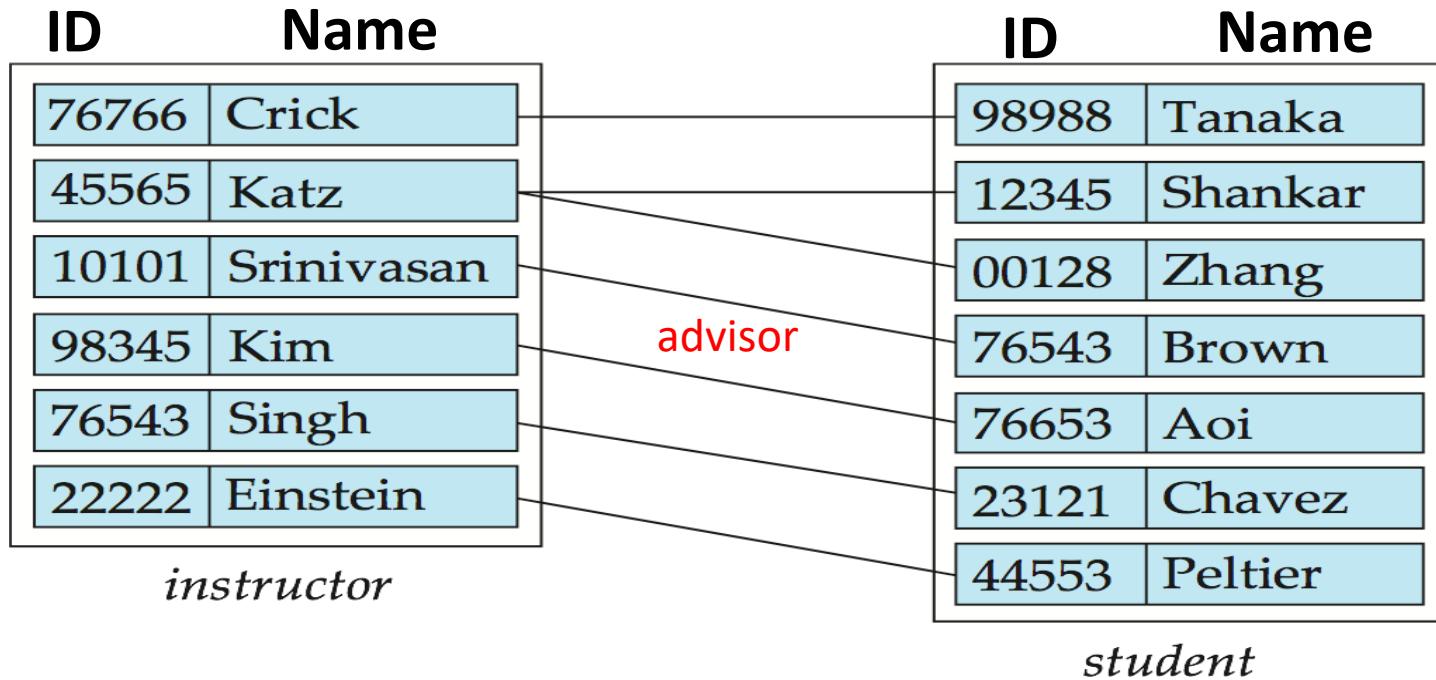
- Suppose we have entity sets
  - *instructor*, with attributes including *dept\_name*
  - *Department* and a relationship
  - *inst\_dept* relating *instructor* and *department*
- If we put Attribute **dept\_name** in entity *instructor* then information (employee works in a department) is redundant since there is an **explicit relationship** *inst\_dept* which relates instructors to departments
  - The **dept\_name** attribute replicates information present in the relationship, and **dept\_name** should be removed from *instructor*
  - **BUT:** when converting ER back to tables, in some cases the attribute gets reintroduced.



# Keys

- A **super key** of an entity set is a set of one or more attributes, allow us to identify uniquely every entity in the entity set.
- A **candidate key** of an entity set is a **minimal super key**.
  - *ID* is candidate key of *instructor*
  - *course\_id* is candidate key of *course*
- Although several **candidate keys** may exist, one of the candidate keys is selected to be the **primary key**.
- Need to consider **semantics of Entity set** in selecting the **primary key** in case of more than one candidate key.
  - Concept is Discussed in Relational Model Chapter

# Keys for Relationship Sets



Relationship set **advisor** ={ (76766,Crick - 98988,Tanaka) , (45565,Katz- 12345,Shankar),  
(45565,Katz-00128,Zhang),(10101,Srinivasan),... (22222, Einstein-44553,Peltier) }

Every element(**Relationship instance**) in Relationship Set is to be identified uniquely, therefore we need a primary key for relationship set too.

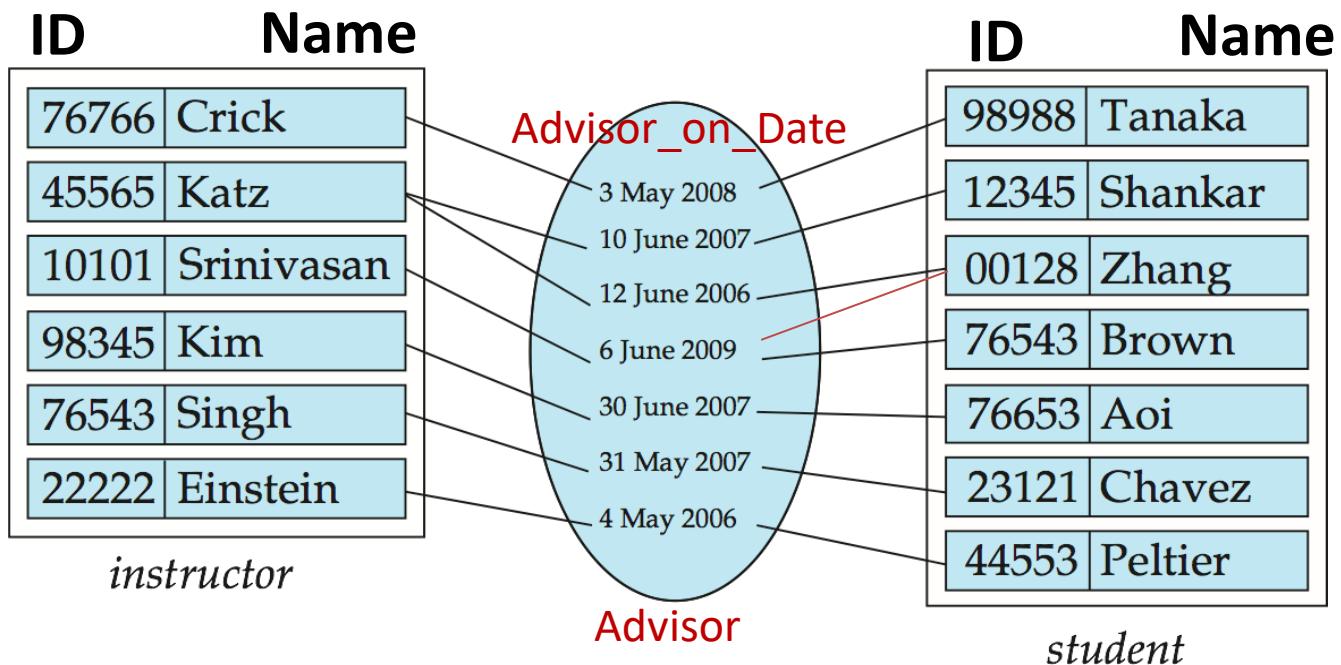
# Keys for Relationship Sets

- The combination of primary keys of the participating entity sets forms the Primary key of a relationship set.
- Let R be a relationship set involving entity sets  $E_1, E_2, \dots, E_n$ . Let **primary-key( $E_i$ )** denote the set of attributes that forms the primary key for entity set  $E_i$ .
- If the relationship set R has **no descriptive attributes** associated with it, then the set of attributes.  
**primary-key( $E_1$ ) U primary-key( $E_2$ ) U...primary-key( $E_n$ )**
- Example:
  - Primary Key for Advisor relationship set is
    - – **Primary Key(Advisor) U Primary Key(Student)**
- i.e **(ID, ID)** , i.e. ID of Instructor & ID of Student is Pkey of Advisor

# Keys for Relationship Sets having Attributes

- If the relationship set R has **descriptive attributes**  $a_1, a_2, \dots, a_m$  associated with it, then the set of attributes- primary-key( $E_1$ )  $\cup$  primary-key( $E_2$ )  $\cup \dots$  primary-key( $E_n$ )  $\cup$   $\{a_1, a_2, \dots, a_m\}$  forms the relationship set R.

Example

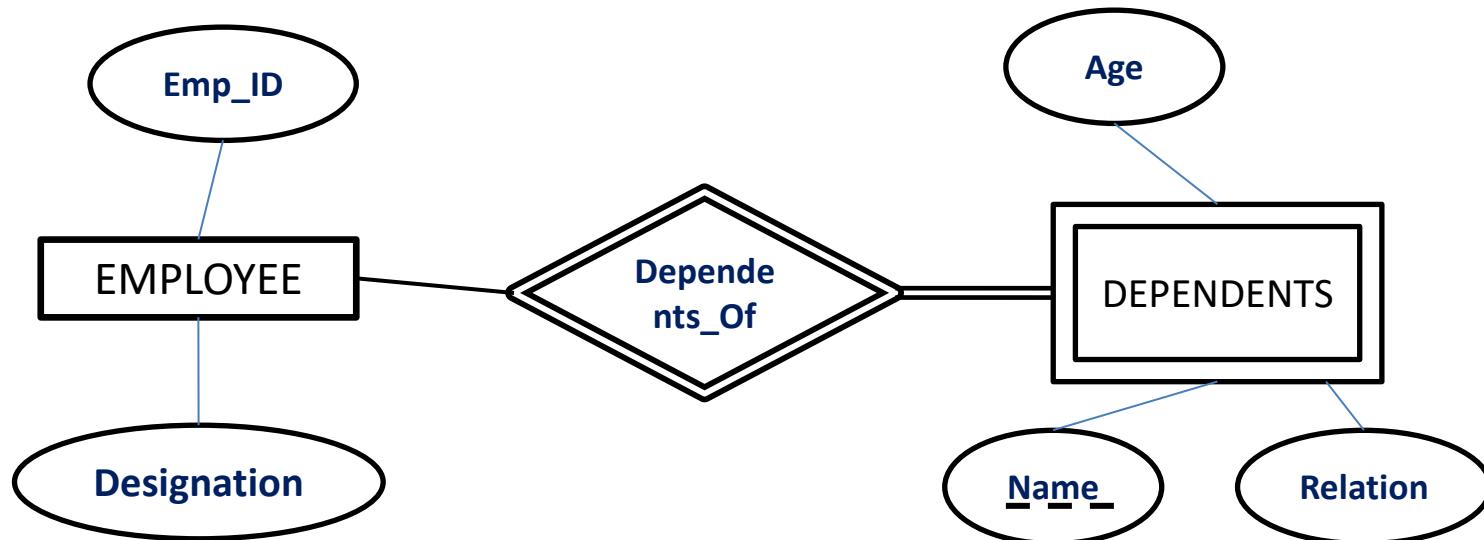


(ID, ID, Advisor\_On\_Date)

# 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 partition, 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**.

# Weak Entity Sets (Cont.)

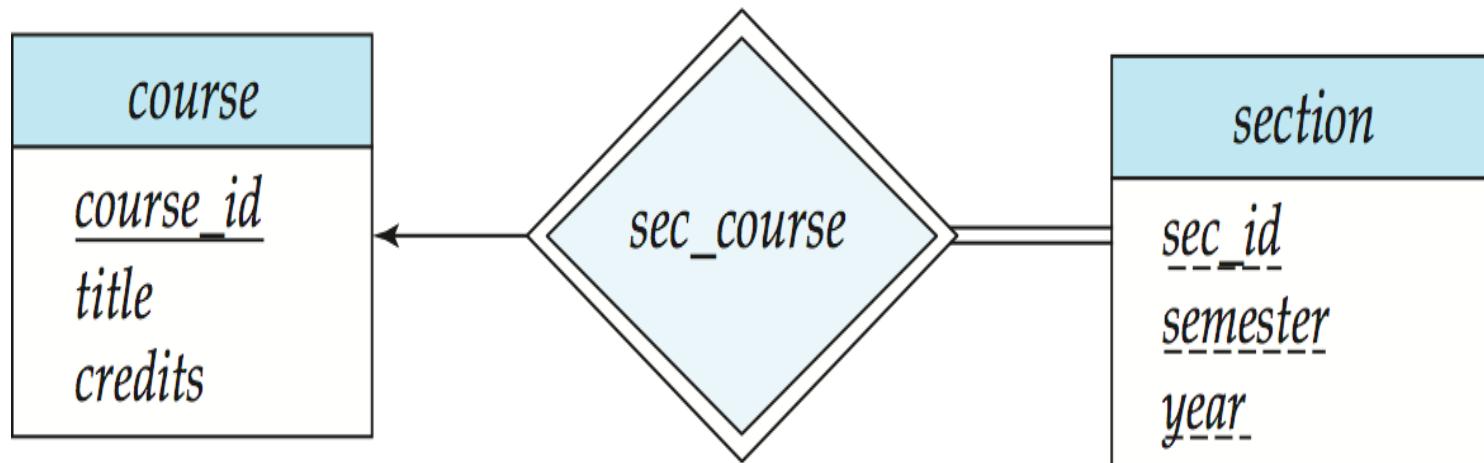


Empno	Designation
100	
101	
102	

Name	Relation	Age
X	Son	12
Y	Daughter	5
Y	Wife	25
P	Son	7

# Weak Entity Sets (Cont.)

- We underline the **discriminator** of a weak entity set with a **dashed line**.
- We put the identifying relationship of a weak entity in a double diamond.
- Primary key for *section* – (*course\_id, sec\_id, semester, year*)



**Partial key/ Discriminator:** Uniquely identifies a section among the set of sections of a particular course

# Weak Entity Sets (Cont.)

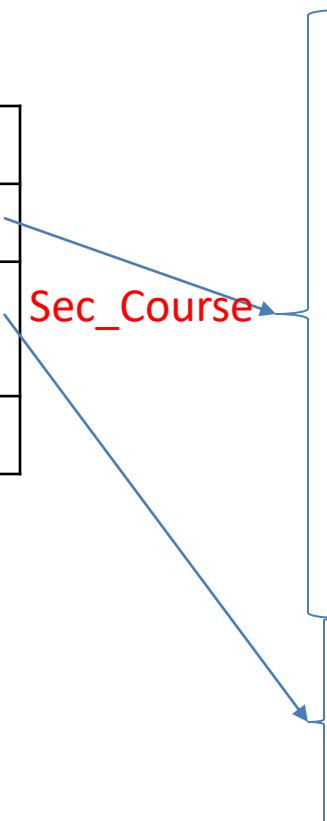
Course

Course_Id	Title	Credits
MCA	Master of..	36
MTech	Info. Security	38
BTech	DSE	36

Section

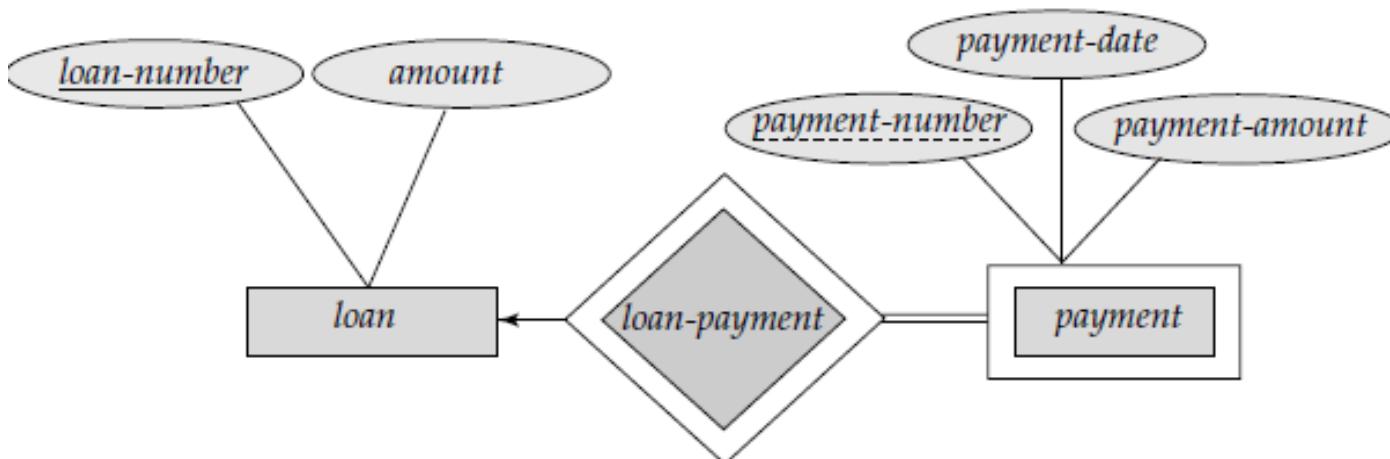
Sec_Id	Semester	Year
A	I	2021
B	I	2021
A	III	2020
B	III	2020
A	II	2020
B	II	2020
A	IV	2019
B	IV	2019
A	I	2019
A	III	2018
...	..	..

Sec\_Course



## 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*.
- Similar idea applies to loan, payment relations given below(how it is represented in scheme ?see next slide)-



<i>loan-number</i>	<i>amount</i>	<i>payment-number</i>	<i>payment-date</i>	<i>payment-amount</i>
L-11	900	53	7 June 2001	125
L-14	1500	69	28 May 2001	500
L-15	1500	22	23 May 2001	300
L-16	1300	58	18 June 2001	135
L-16		5	10 May 2001	50
L-17	1000	6	7 June 2001	50
L-17		7	17 June 2001	100
L-23	2000	11	17 May 2001	75
L-93	500	103	3 June 2001	900
L-93		104	13 June 2001	200

Loan Entity Type

Payment Entity Type – Weak Entity

<i>loan-number</i>	<i>payment-number</i>	<i>payment-date</i>	<i>payment-amount</i>
L-11	53	7 June 2001	125
L-14	69	28 May 2001	500
L-15	22	23 May 2001	300
L-16	58	18 June 2001	135
L-17	5	10 May 2001	50
L-17	6	7 June 2001	50
L-17	7	17 June 2001	100
L-23	11	17 May 2001	75
L-93	103	3 June 2001	900
L-93	104	13 June 2001	200

Payment Entity Type can be Identified with respect Loan using PK of Loan

# Information to be fetched from Requirements

- Entities
  - Strong
    - Attributes-Primary key, Composite/ Simple, Multivalued/Single valued, Derived.
  - Weak
    - Partial key
- Relationship Between Entities
  - Relationship Attributes
  - Name of relationship
  - Cardinality Constraint
    - **1-1, 1-M, M-M**
    - Cardinality Limits
  - Participation
    - Total/Partial

## Example

An Institute want to keep track of information about Funded Projects, Agencies which are Funding them and Faculties who work on those Projects.

The institute is comprised of several departments such as – DS&MCA,CS,IT ,MECH, EEE etc.. and each department has a Department Number such as D1,D2,.. used to identify each department. Many faculties work in every department . Each faculty is identified by a unique Employee Number. Information about each faculty we need is employee name, Qualification, Research-domain. Each Departments may have many funded research projects. Information about these projects such as an unique Project-ID, Title, Fund-Received, Duration. Each of these projects are funded by one or more funding agencies such as – MHRD,DSR, DST,BARC etc.. We also need to record information about Funded Funding agencies funding the projects such as- Grant\_ord\_No, Agency-Name, Contact-Person, Email, Phone, Total\_Grant,Year-of-Grant. An agency may fund multiple project and a project may also receive grant from multiple agencies.

Model above data requirements using ER modeling.

**What are Entities here ?**

**Department**

**What are attributes & sample entities here.**

**Deptno, Dname**

D1	CS
D2	IT

**Empno, Ename, Qualification, Research-Domain**

**Faculty**

101	Raj	MTech	Data-Mining
102	Vinay	Mtech,PhD	Network Eng
106	Manu	MCA, PhD	AI

**Project-ID, Title, Fund-Received, Duration**

**Projects**

P1	XYXX	200K	2
P2	GHKL	500K	3
P3	FGHK		

**Agency**

**Grant\_order\_No, Agency-Name, Contact-Person, Email, Phone, Total\_Grant, Year-of-Grant**

MH17-1	MHRD	RamRao	ram@gmail.com	78998667	2000K	2017
MH18-5	MHRD	Vijay	Vij@ymail.com	89532689	5000K	2018
DST17-3	DST	Ravi	Ravi@gmail.com	99644775	3500K	2017

# Entities

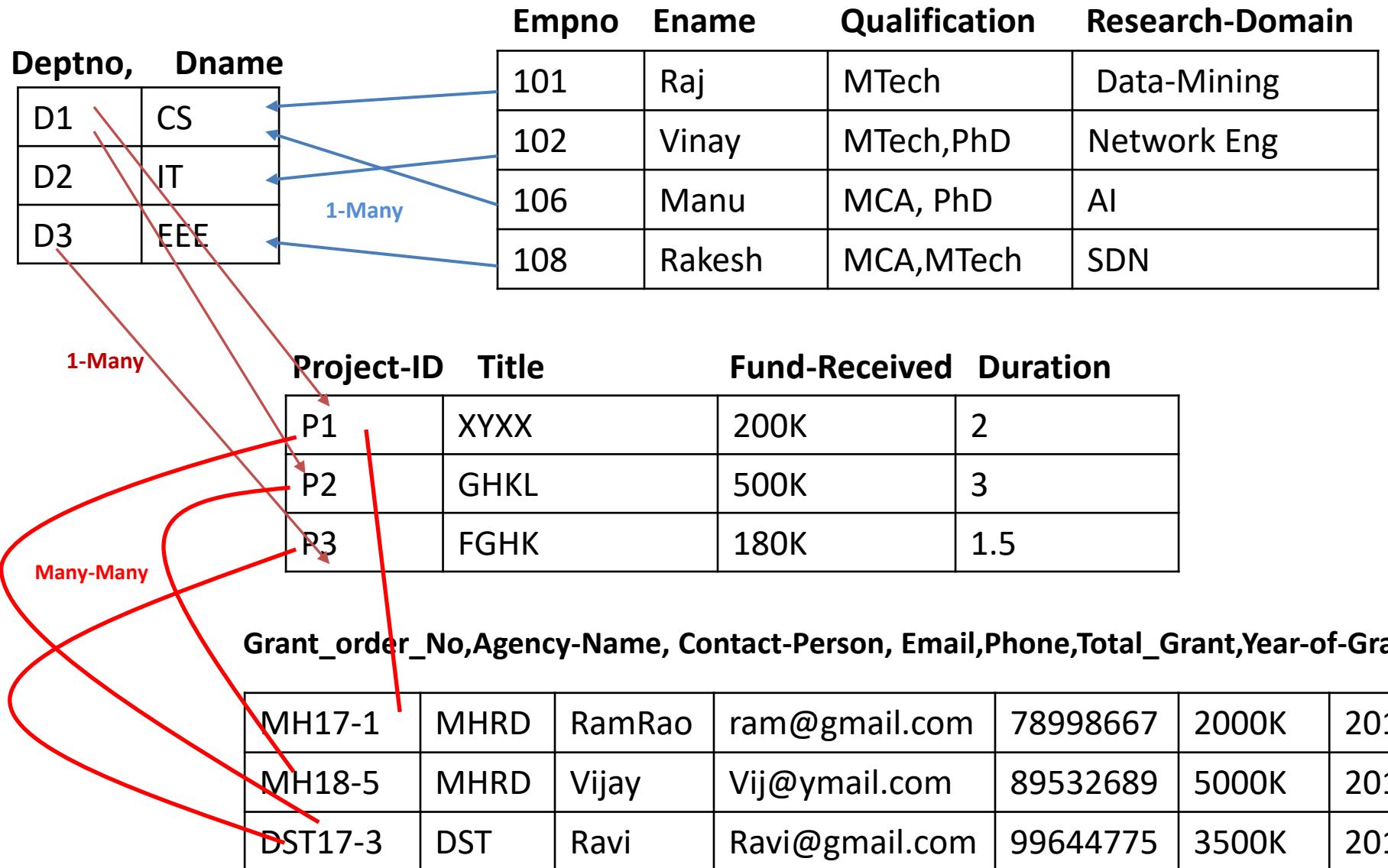
DEPARTMENT
<u>DeptNo</u>
DName

PROJECT
<u>Proj-ID</u>
Prj-Name
Duration
Fund_Received

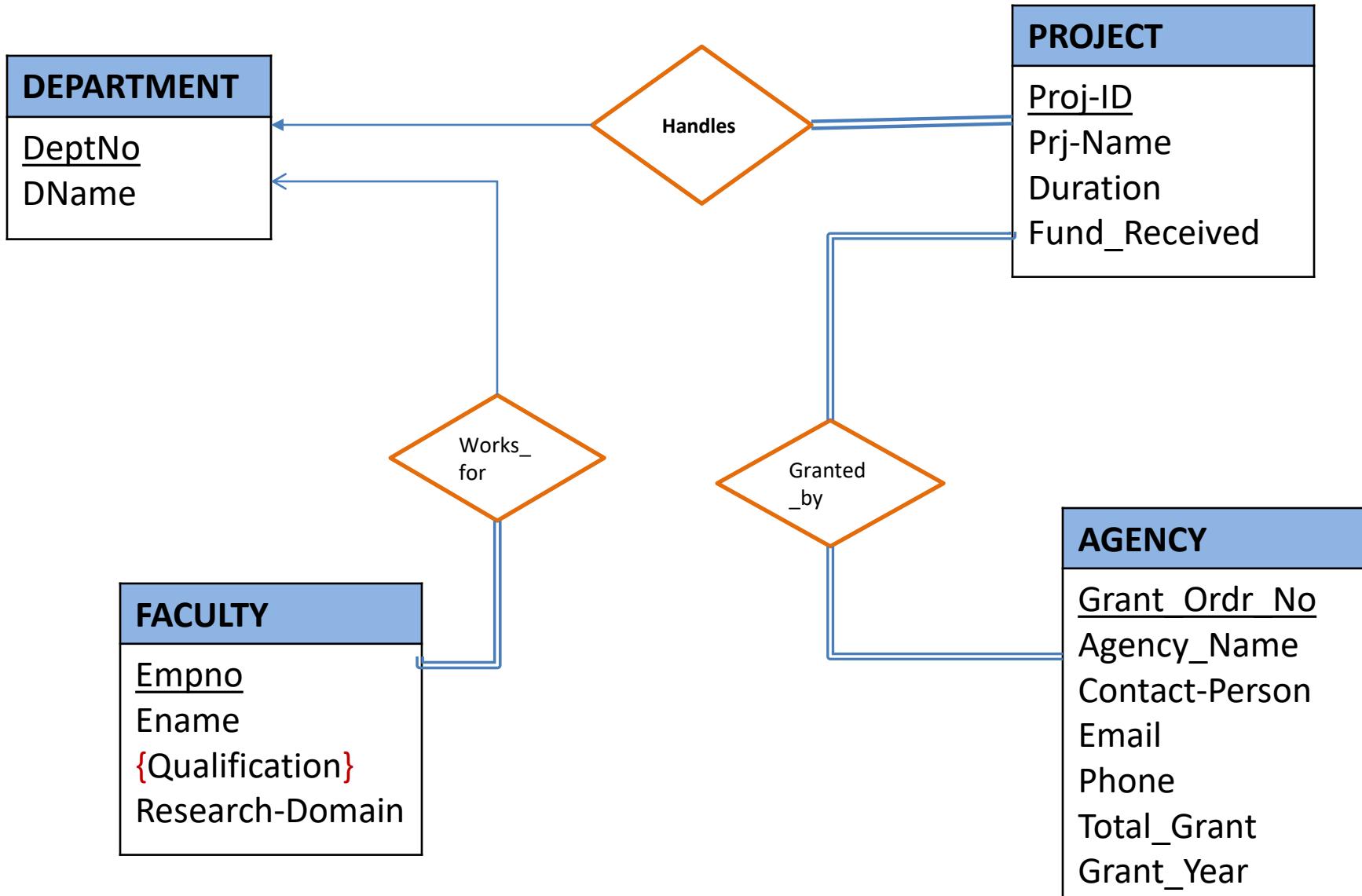
FACULTY
<u>Empno</u>
Ename
{Qualification}
Research-Domain

AGENCY
<u>Grant_Ord_No</u>
Agency_Name
Contact-Person
Email
Phone
Total_Grant
Grant_Year

# Entities and Relationship



# Entities & Relationship



# **Reduction to Relational Schemas**

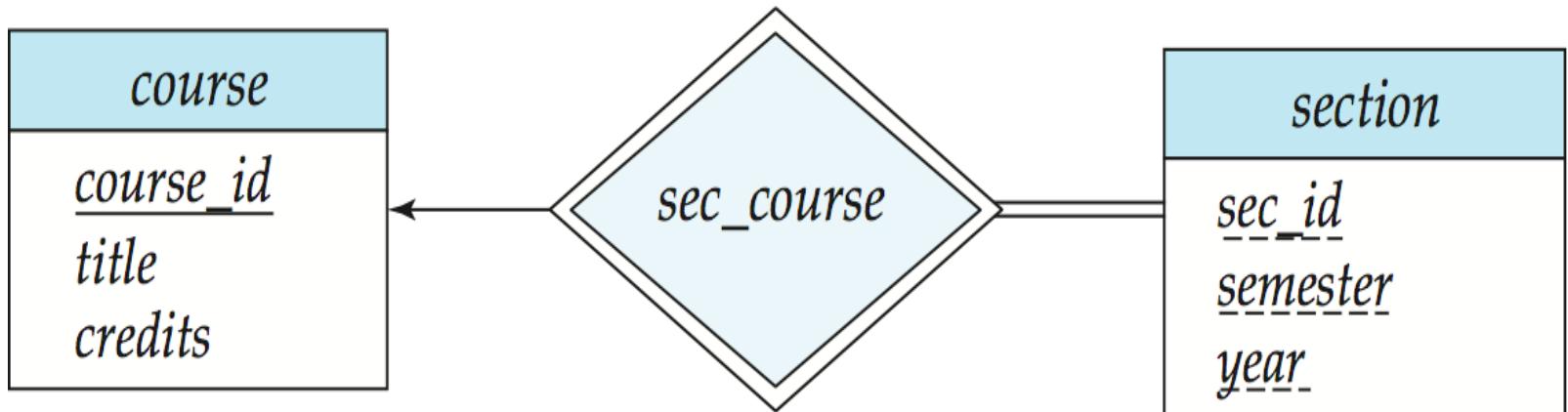
# Reduction to Relation Schemas

- 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**.

# Representing Entity Sets With Simple Attributes

- A **strong entity set** reduces to a schema with the same attributes  
*Course(Course\_ID, title, tot\_cred)*
- A **weak entity set** becomes a table that includes a column for the **primary key of the identifying strong entity set**.

*section ( course\_id, sec\_id, sem, year )*



**Note:** schemas *sec\_course(course\_id, sec\_id, sem, year)* is not represented because *sec\_course* schema **is redundant** in *Section (course\_id, sec\_id, sem, year)* schema

# Redundancy in Schema Representation

The schema corresponding to a **relationship set linking a weak entity set to its identifying strong entity set** is **redundant**.

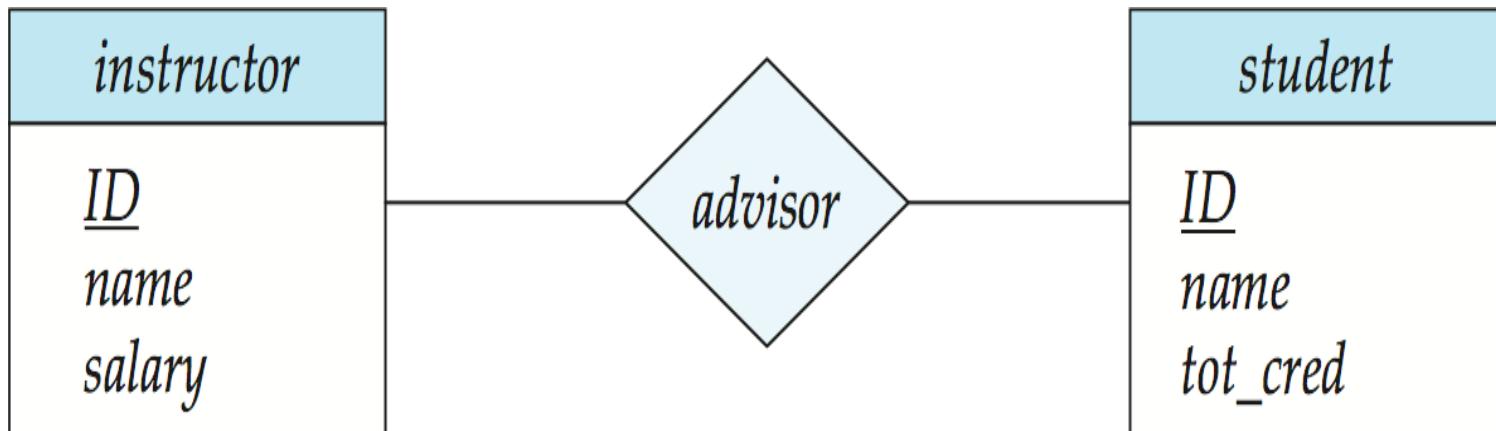
**Example:** The *section* schema already contains the attributes that would appear in the *sec\_course* schema

# Representing Relationship Sets Many-Many

- 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)**

**Instructor( ID, Name, Salary) & Student(ID, Name, tot\_cred)**



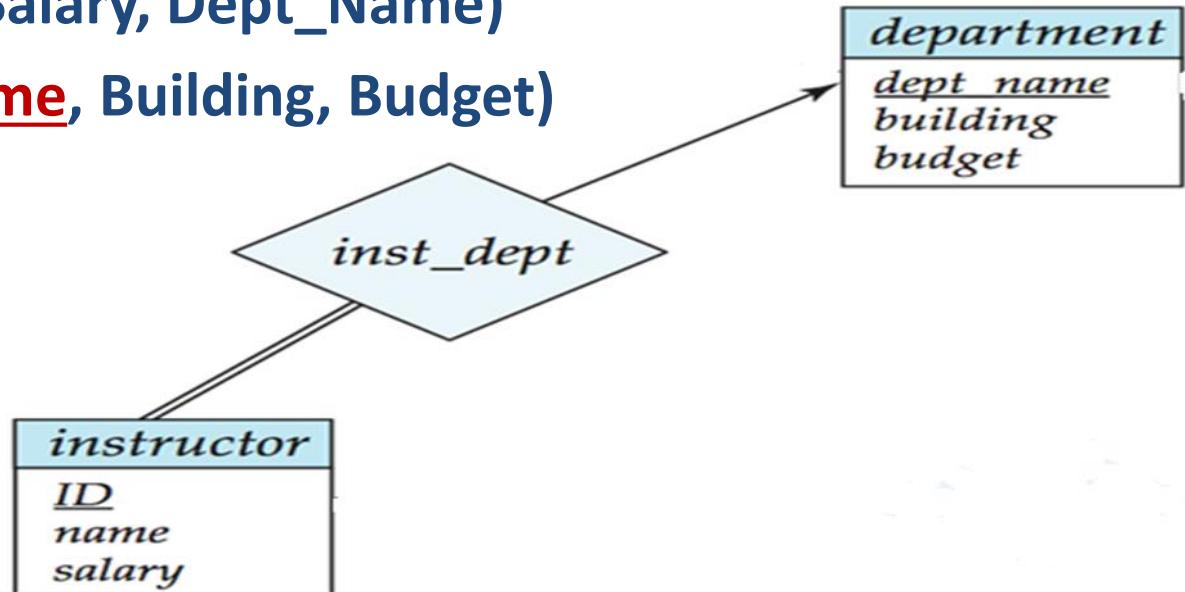
# Representing Relationship sets- Many-1/1-Many

- 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
- Example: Instead of creating a schema for relationship set *inst\_dept*, add an attribute *dept\_name* to the schema arising from entity set *instructor*

Instructor( ID, Name, Salary, Dept\_Name)

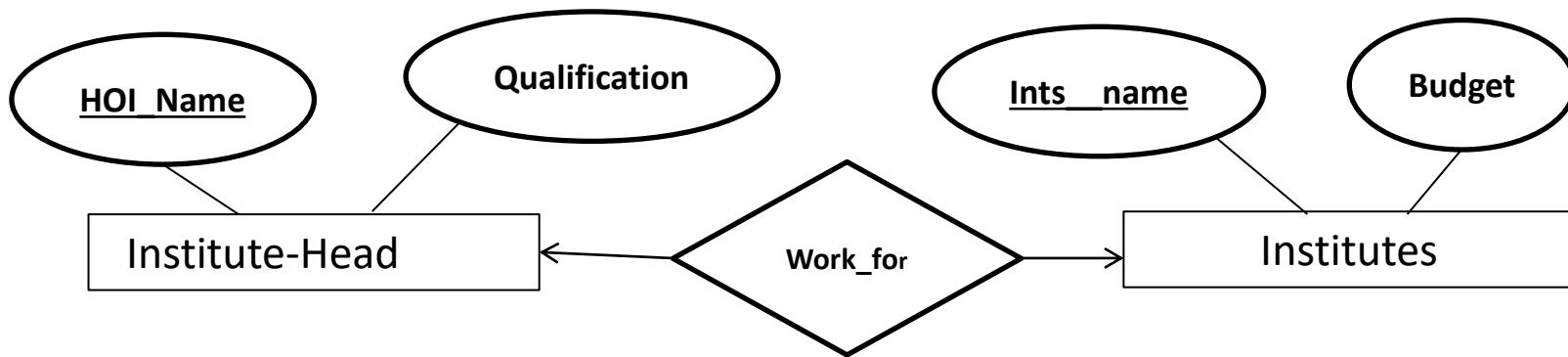
Department(Dept\_Name, Building, Budget)

Dept\_Name in Instructor is taken as Foreign key referencing Department.



# Representing Relationship sets- 1 to 1

- For one-to-one relationship sets, either side P.key can be chosen to act as F.key at other side

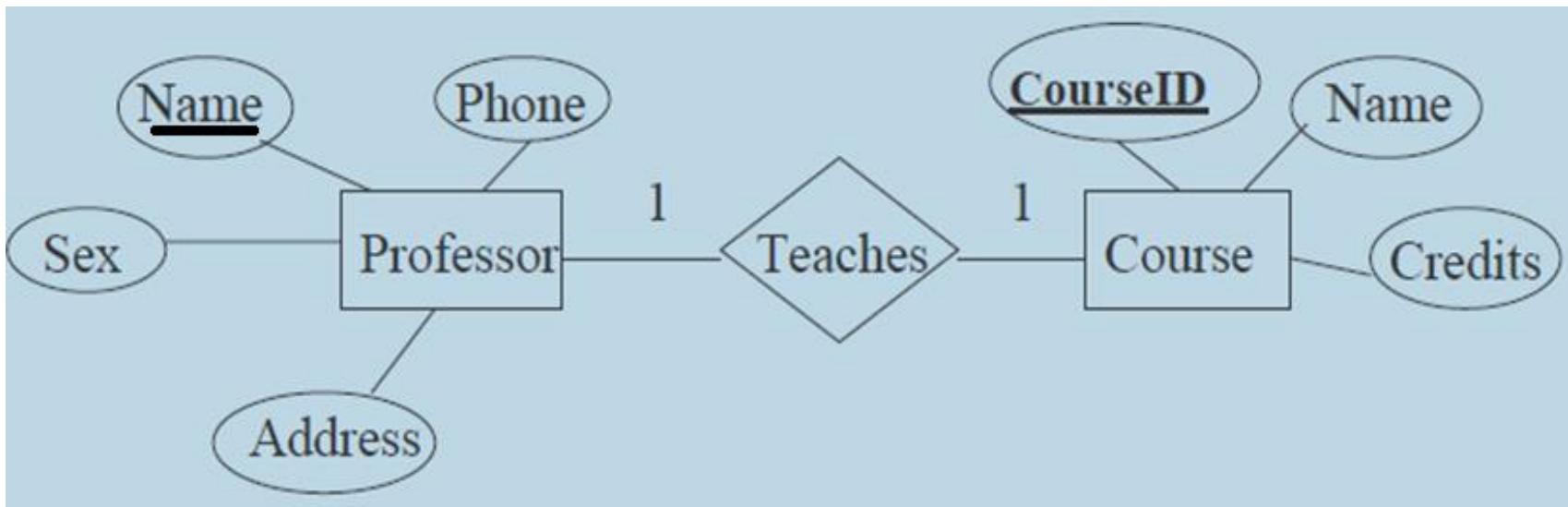


Institute\_Head(HOI\_Name, Qualification )

Institute(Inst\_Name, Budget, HOI\_Name)

**HOI\_Name** in Institute is taken as **Foreign key** referencing **Institute\_Head**.

## Reduce the Following ER Diagrams into Relational Schema.

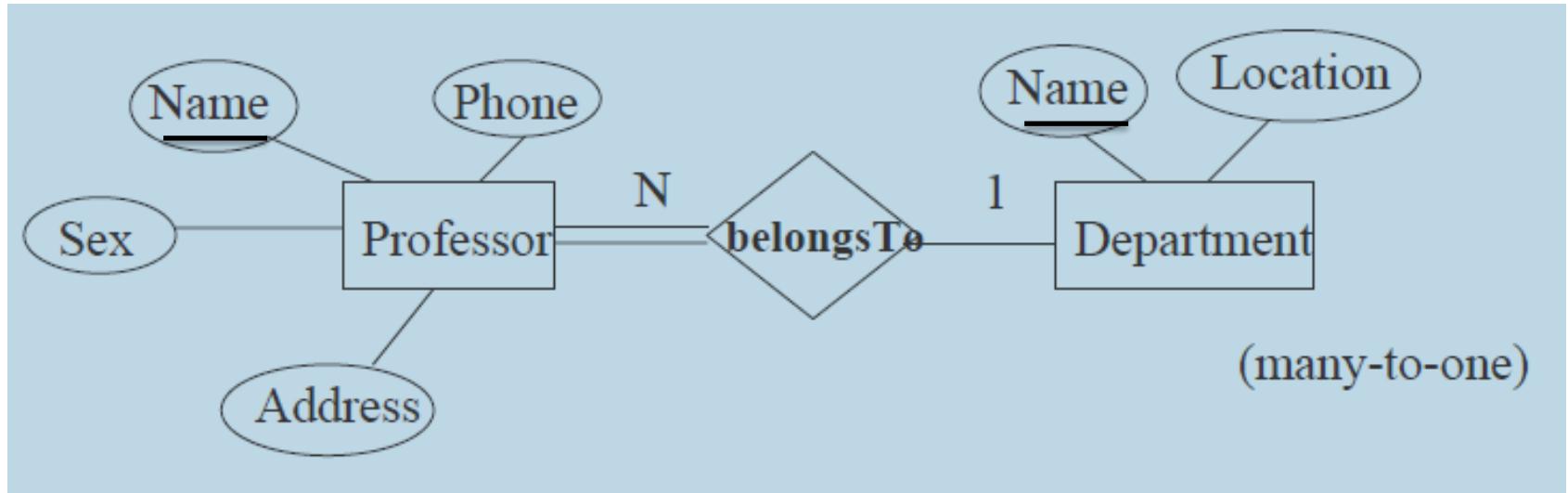


Professor(Name, Phone, Address, Sex, **CourseID**)

Course(CourseID, Name, Credits)

Professor.CourseID is foreign key referencing Course.CourseID

# Reduce the Following ER Diagrams into Relational Schema.



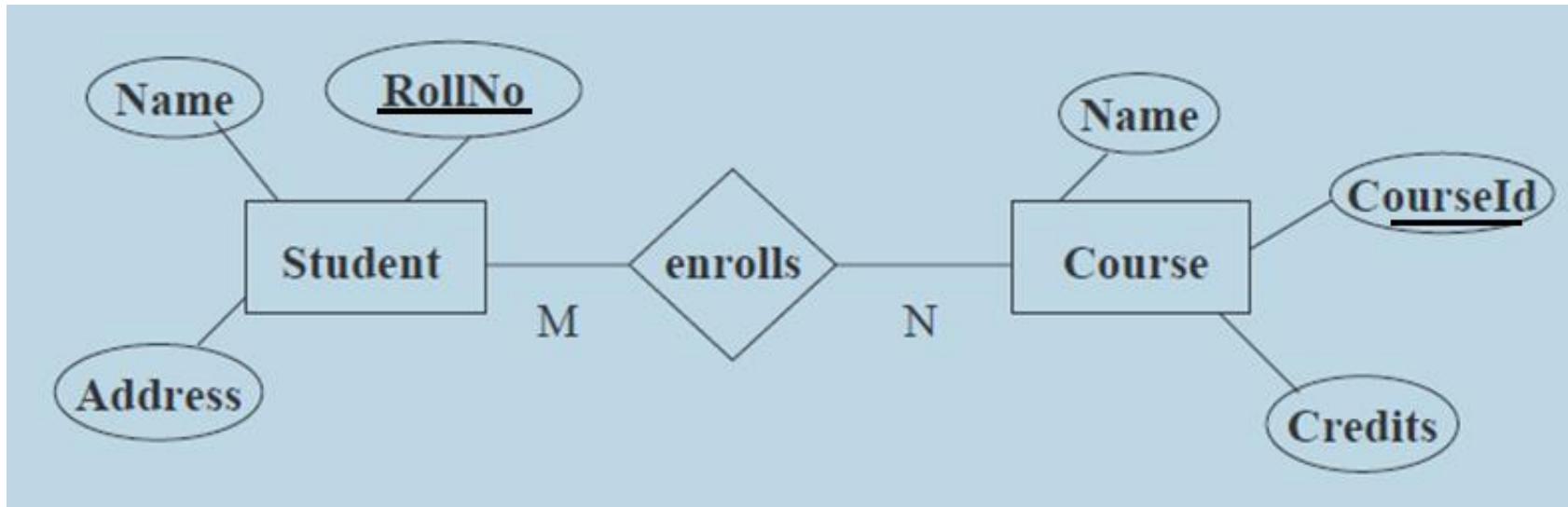
Requirement collected says-A department may have many Professor

Professor(Name, Phone, Address, Sex, **Dep\_Name**)

Department(Name, Location)

Dep\_Name is foreign key referencing Department. Name

# Reduce the Following ER Diagrams into Relational Schema.



Assume, Requirement says- **A course is enrolled by multiple students and a student may also enroll to multiple course**

Assume **Student.Rollno** and **Course.CourseId** is Primary Key in Student and Course entity types respectively.

**Student(Name, RollNo, Address)**

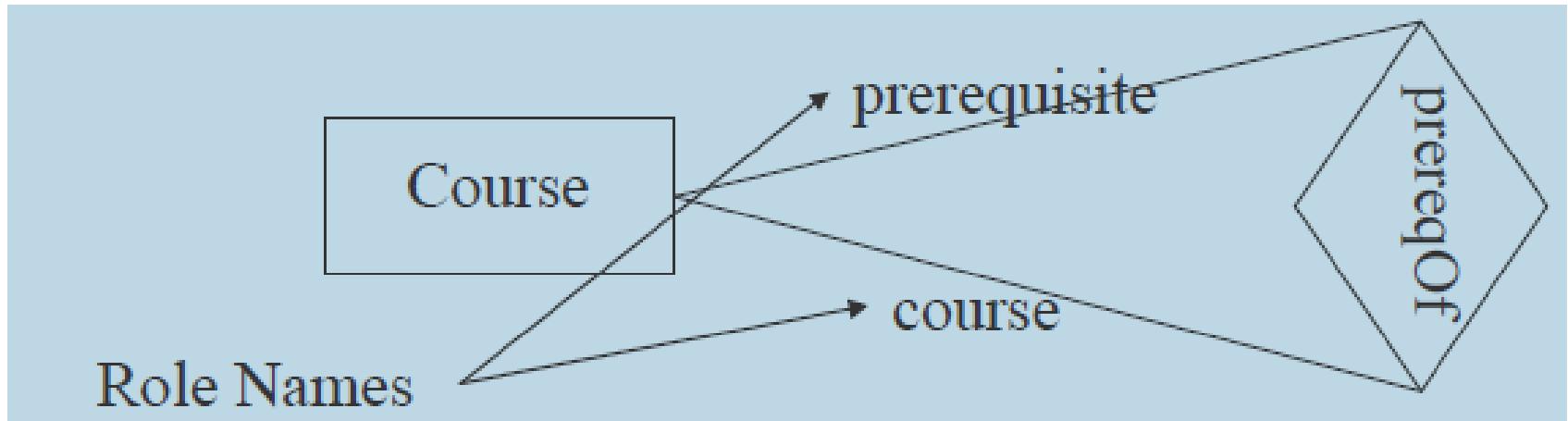
**Course(Name, CourseId, Credits)**

**Enrolls(RollN,CourseId)**

# Reduce the Following ER Diagrams into Relational Schema.- Recursive Relationship

Assume that the Prerequisite for a Course is another Course.

Assume COURSE has attributes **CourseId**, **Course\_name**, **Credits**



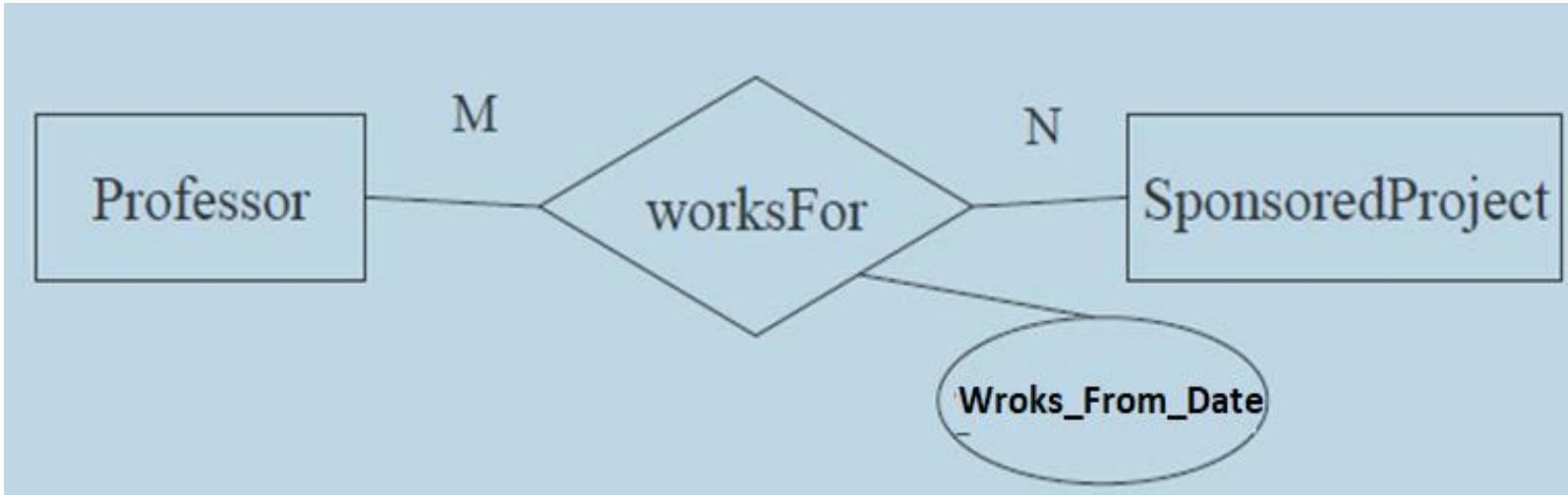
Course(CourseId, Course\_Name, Credits, Pre\_CourseId)

**Note:** It is **recursive** relationship

Pre\_CourseId (**F.key**) Referencing **CourseId** (**P.Key**)

# Reduce the Following ER Diagrams into Relational Schema –with Descriptive Attribute

Assume Professor has attributes – Empno, Name ,Address  
and SponsoredProject has attributes – Prj\_Id, Pname, Duration)  
Professors start working on different project from different dates



Professor(Empno, Name ,Address)

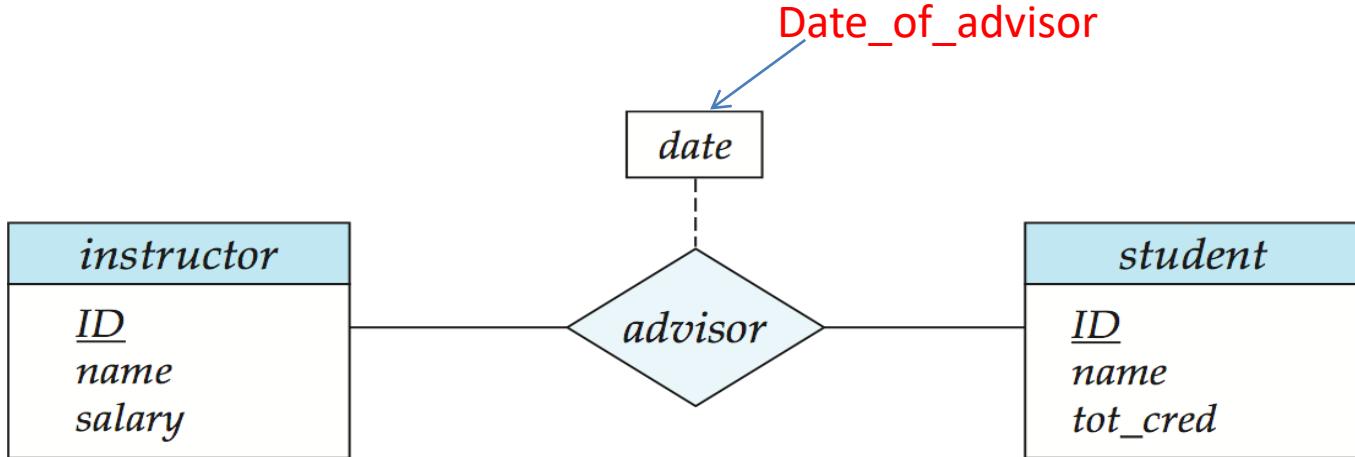
SponsoredProject(Prj\_Id, Name, Duration)

Works\_for(Empno, Prj\_Id, Works\_From\_Date)

# Reduce the Following ER Diagrams into Relational Schema –with Descriptive Attribute

Keys for Relationship Sets having Attributes																																													
	<ul style="list-style-type: none"> <li>If the relationship set R has descriptive attributes <math>a_1, a_2, \dots, a_n</math> associated with it, then the set of attributes <math>\text{primary-key}(E_1) \cup \text{primary-key}(E_2) \cup \dots \cup \text{primary-key}(E_n) \cup \{a_1, a_2, \dots, a_n\}</math> forms the relationship set R.</li> </ul>																																												
<b>Example</b>																																													
	<table border="1"> <thead> <tr> <th>ID</th> <th>Name</th> <th>Advisor_ID</th> <th>Date</th> </tr> </thead> <tbody> <tr><td>76766</td><td>Crick</td><td>12345</td><td>2010-01-01</td></tr> <tr><td>47655</td><td>Katia</td><td>12345</td><td>2010-01-01</td></tr> <tr><td>10101</td><td>Chandrasekhar</td><td>12345</td><td>2010-01-01</td></tr> <tr><td>88051</td><td>Kim</td><td>12345</td><td>2010-01-01</td></tr> <tr><td>76943</td><td>Singh</td><td>12345</td><td>2010-01-01</td></tr> <tr><td>29222</td><td>Emissions</td><td>12345</td><td>2010-01-01</td></tr> </tbody> </table> <table border="1"> <thead> <tr> <th>ID</th> <th>Name</th> </tr> </thead> <tbody> <tr><td>18848</td><td>Tanaka</td></tr> <tr><td>12345</td><td>Chandrasekhar</td></tr> <tr><td>112345</td><td>Zheng</td></tr> <tr><td>178451</td><td>Patton</td></tr> <tr><td>769431</td><td>Aoi</td></tr> <tr><td>20121</td><td>Chavez</td></tr> <tr><td>44955</td><td>Potter</td></tr> </tbody> </table>	ID	Name	Advisor_ID	Date	76766	Crick	12345	2010-01-01	47655	Katia	12345	2010-01-01	10101	Chandrasekhar	12345	2010-01-01	88051	Kim	12345	2010-01-01	76943	Singh	12345	2010-01-01	29222	Emissions	12345	2010-01-01	ID	Name	18848	Tanaka	12345	Chandrasekhar	112345	Zheng	178451	Patton	769431	Aoi	20121	Chavez	44955	Potter
ID	Name	Advisor_ID	Date																																										
76766	Crick	12345	2010-01-01																																										
47655	Katia	12345	2010-01-01																																										
10101	Chandrasekhar	12345	2010-01-01																																										
88051	Kim	12345	2010-01-01																																										
76943	Singh	12345	2010-01-01																																										
29222	Emissions	12345	2010-01-01																																										
ID	Name																																												
18848	Tanaka																																												
12345	Chandrasekhar																																												
112345	Zheng																																												
178451	Patton																																												
769431	Aoi																																												
20121	Chavez																																												
44955	Potter																																												
	(ID, ID, Advisor_ID, Date)																																												

Assume the requirement is- many Instructors can be advisor to multiple students at different dates.



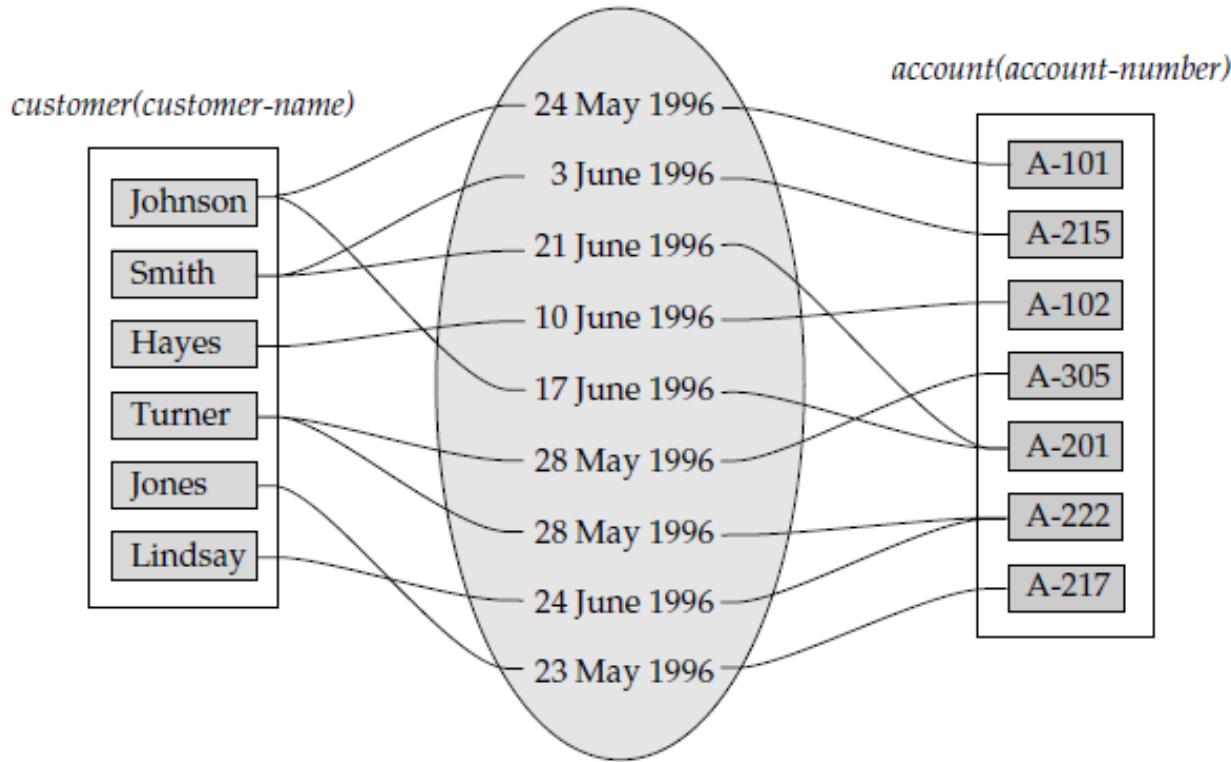
Instructor(ID, name, Salary)

Student(ID, name, tot\_cred)

Advisor(I.Id, S.Id, Date\_of\_advisor)

## Relation –with Descriptive Attribute

*depositor(access-date)*



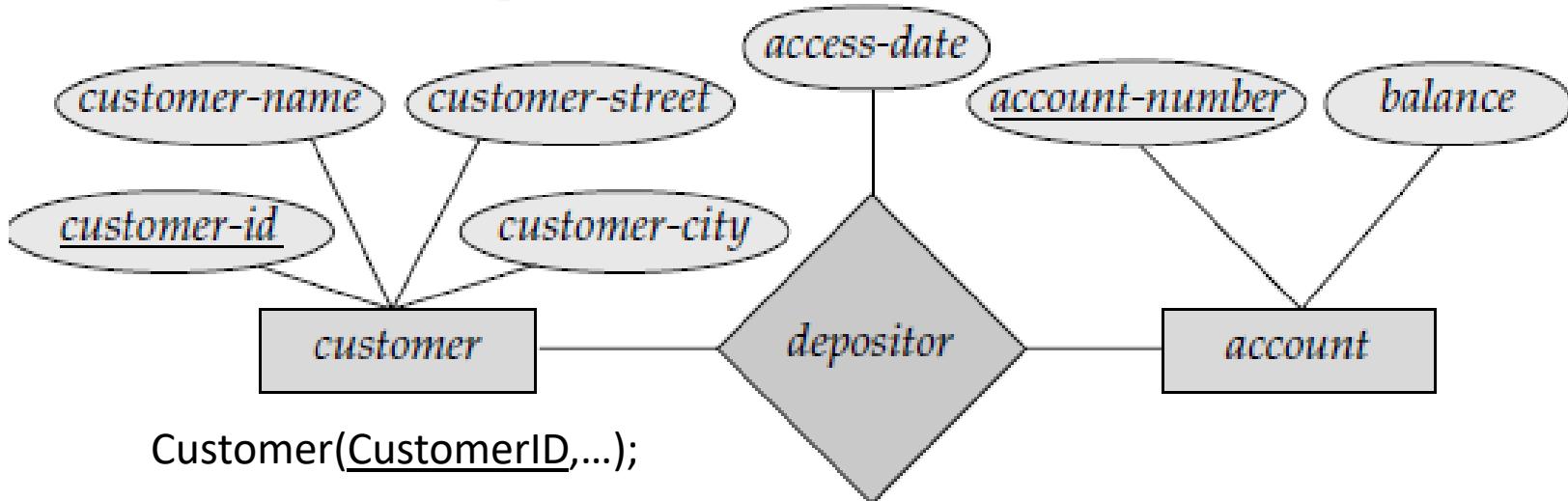
Customer and Account are having many-to-many relationship and the *access\_date* descriptive attribute attached to the relationship set *depositor* to specify the most recent date on which a customer accessed that account.

However if we want to record **all dates** when customers have accessed their accounts then scenario will be different while converting into schema.

In that case relationships like –

(Smith,21-June-1996,A-201) and (Smith,21-June-1996,A-215),(Smith,21-June-2006,A-201) may exist

## ER Diagram & Schema –with Descriptive Attribute



Customer(CustomerID,...);

Account(Account Number, Balance)

Depositor(Account Number,CustomerID,Access Date)

However if we want to record all dates when customers have accessed their accounts then scenario will be different while converting into schema.

In that case relationships like –

**(Smith,21-June-1996,A-201)** and **(Smith,21-June-1996,A-215)**, **(Smith,21-June-2006,A-201)** may exist.

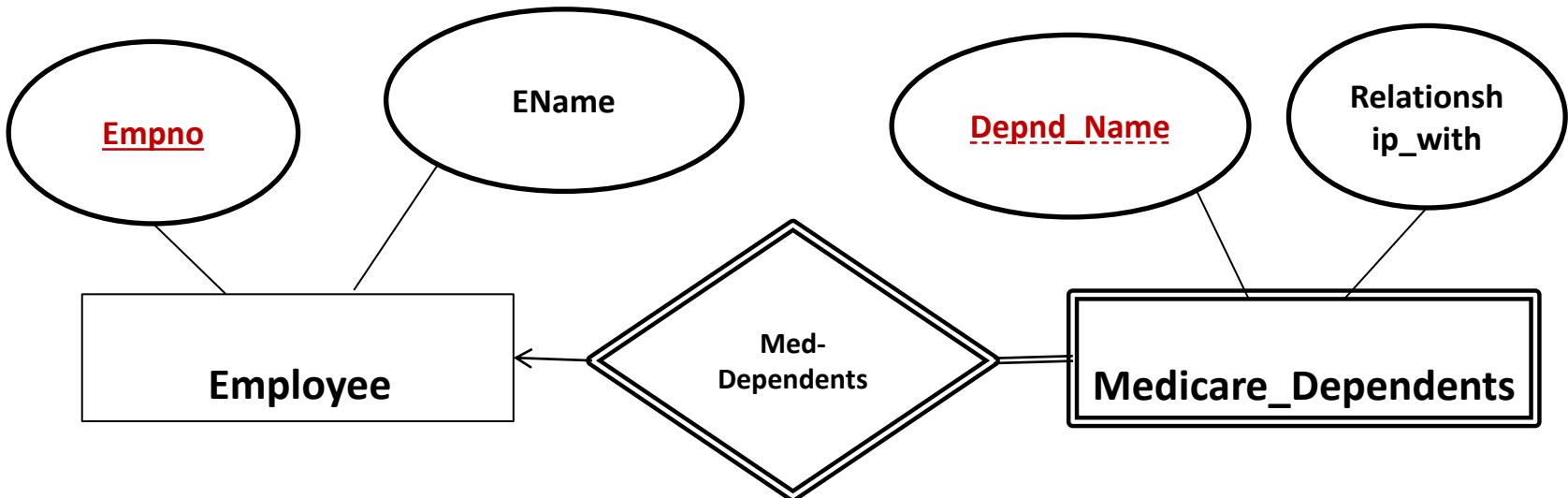
Now Access\_Date also will be part of Primary key

Customer(CustomerID,...);

Account(Account Number, Balance)

Depositor(Account Number, CustomerID, Access Date)

## Reduce the Following ER Diagrams into Relational Schema- Weak Entity.



Employee(Empno,Ename)

Medicare\_Dependents(Empno,Depnd\_Name, Relationship\_with)

Empno in Medicare\_Dependents is Foreign key Referencing Empno in Employee.

# Representing Composite Attributes in Schema

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

- 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.*
- *instructor* schema is
  - **Instructor ( ID,**  
*first\_name, middle\_initial, last\_name,*  
*street\_number, street\_name,*  
*apt\_number, city, state, zip\_code,*  
***date\_of\_birth)***

# Representing Multivalued Attributes in Schema

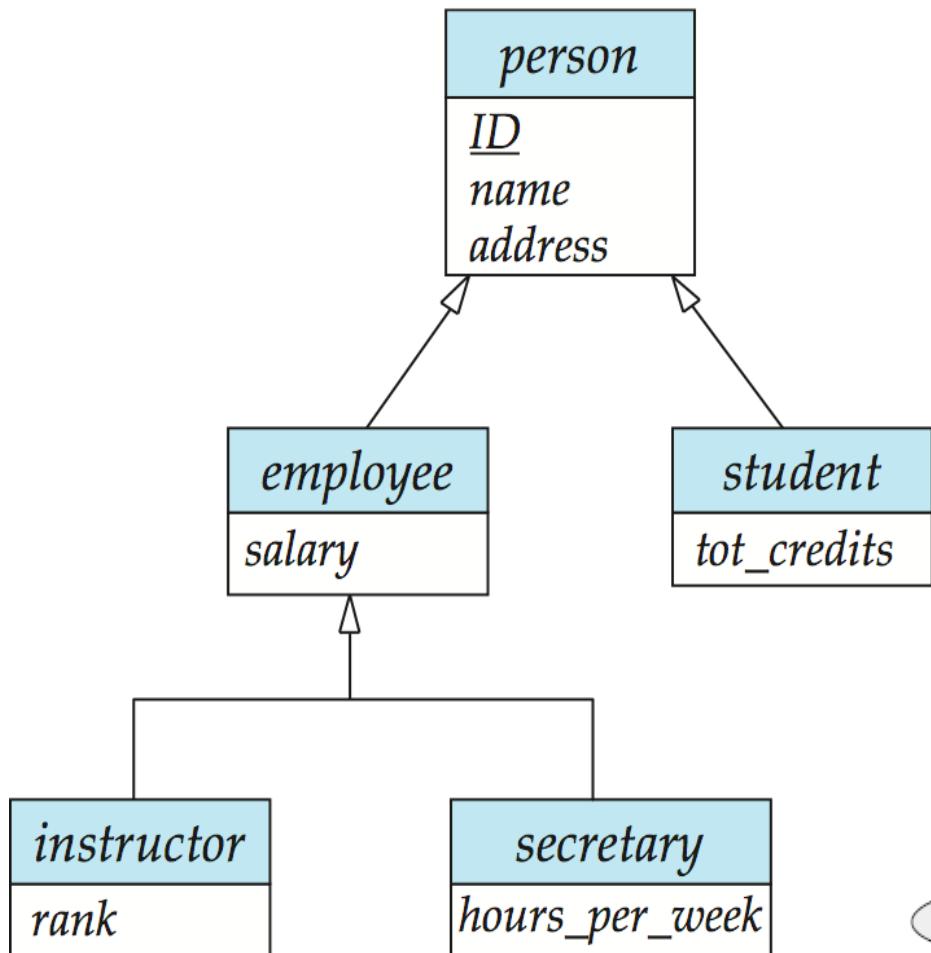
- A multivalued attribute  $M$  of an entity  $E$  is represented by a separate schema  $EM$ 
  - Schema  $EM$  has attributes corresponding to the primary key of  $E$  and an attribute corresponding to multivalued attribute  $M$
  - Example: Multivalued attribute  $phone\_number$  of  $instructor$  is represented by a schema:  
 $inst\_phone$  ( $ID$ ,  $phone\_number$ )
  - 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(multi valued):  
(22222, 456-7890) and (22222, 123-4567)

# Extended E-R Features

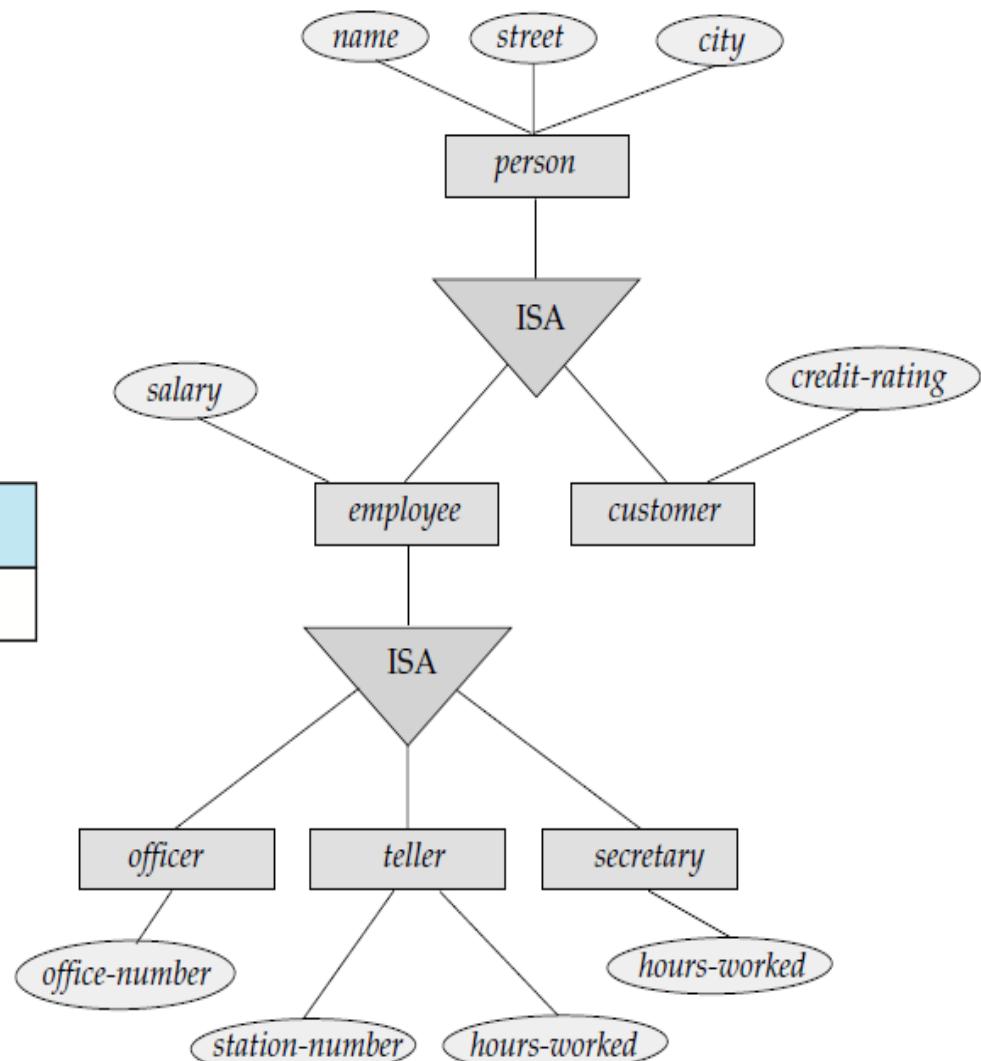
# Extended E-R Features: Specialization

- Specialization is a Top-down design process in which we designate subgroupings within an entity set.
- These subgroupings become lower-level entity sets that have attributes that do not apply to the higher-level entity set.
- Attribute inheritance – A lower-level entity set inherits all the attributes and relationship participation of the higher-level entity set to which it is linked.
- Depicted by a *triangle* component labeled ISA  
(E.g. *instructor* “is a” *person*).
- It can also be referred as a *superclass-subclass* relationship

# Specialization / Generalization Example



(6<sup>th</sup> Edition)



(4<sup>th</sup> Edition)

# Design Constraints on a Specialization/ Generalization

- Constraint on whether or not entities may belong to more than one lower-level entity set within a single generalization.

## – Disjoint

- An higher level entity **can belong to only one lower-level entity set**
- Noted in E-R diagram by having multiple lower-level entity sets link to the same triangle see Accounts , Savings and Checking entities in **next slide**.

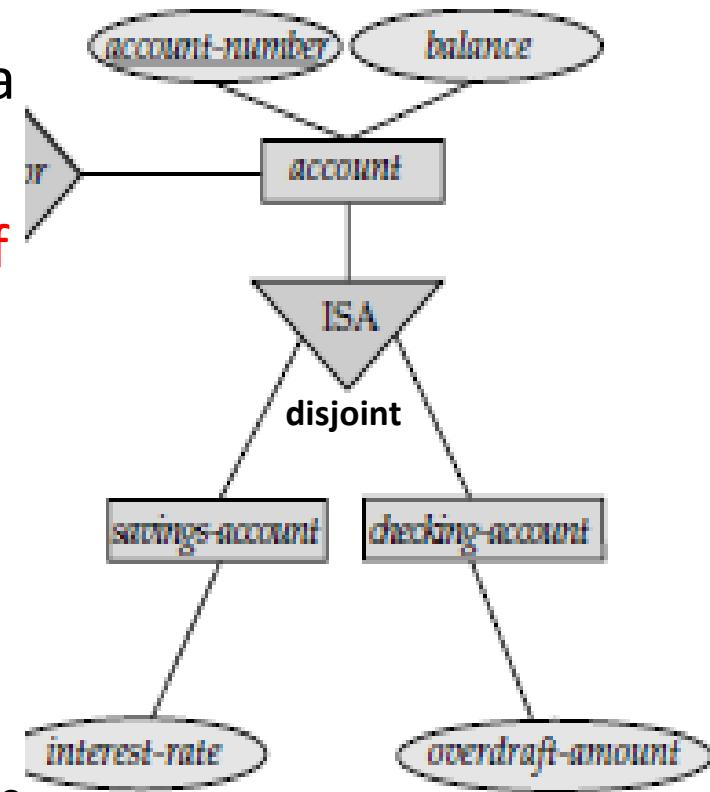
## – Overlapping

- an entity **can belong to more than one lower-level entity set.**
- In a **Person Entity set** is Specialized into Sub classes- **Student** entity set & **Teaching\_Assistants**.
- –**Some persons are student only and some persons may be students as well as Teaching Assistants**

# Design Constraints on a Specialization/Generalization (Cont.)

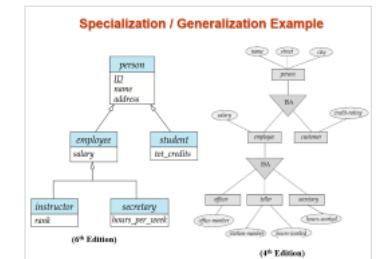
- **Completeness constraint** - specifies whether or not an entity in the higher-level entity set **must belong to at least one** of the lower-level entity sets within a generalization.
  - **total** : an entity **must belong to one of the lower-level entity sets**

Account either have to be savings or checking account



- **partial**: Some higher-level entities may not belong to any lower-level entity set..

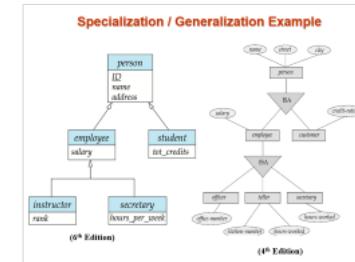
Assume that, **after 3 months (or may be after passing some qualifying exam)** of employment, bank employees are assigned to one of four work teams. Therefore before 3 months there will be some employee entities in the top not belonging to any work team lower entities.



# Representing Specialization via Schemas

- Method 1:
  - Form a schema for the higher-level entity
  - Form a schema for each lower-level entity sets, include primary key of higher-level entity set and local attributes

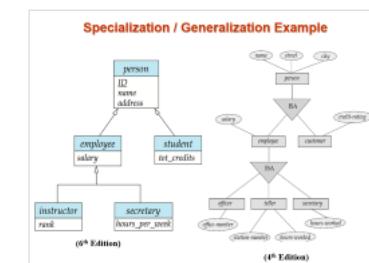
schema	attributes
<i>person</i>	<i>ID, name, street, city</i>
<i>student</i>	<i>ID, tot_cred</i>
<i>employee</i>	<i>ID, salary</i>



- Drawback: getting information about, an *employee* requires accessing two relations, the one corresponding to the low-level schema(**employee**) and the one corresponding to the high-level schema(**person**)

# Representing Specialization as Schemas (Cont.)

- Method 2:
  - Form a schema for each entity set with all local and inherited attributes
  - If specialization is total, the schema for the generalized entity set (*person*) not required to store information
    - Can be defined as a “view” relation containing union of specialization relations
  - Drawback: *name*, *street* and *city* may be stored redundantly for people who are both students and employees



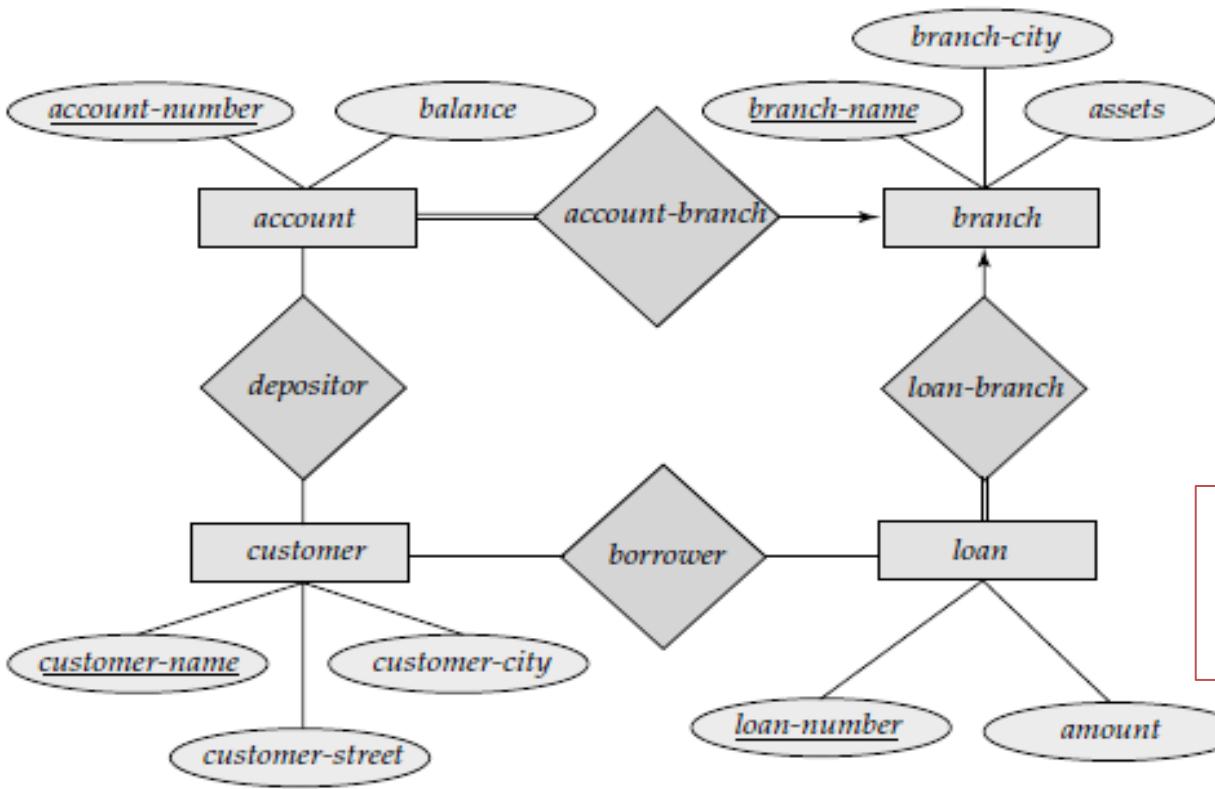
## Draw an ER diagram for the following Requirements

Consider a Banking system, which stores information about its different branches at different locations. Each branch has a unique branch name, location and assets (in Rupees).

Each branch maintains many Savings accounts of different customers. System is interested to record SB account number and balance amount of each savings account.

Each customer may have many accounts and an account may be owned jointly by multiple Customers. Customer information such as-unique Customer\_Name, city and street need to be stored in the system.

Each branch also gives loans to customers. About Loans we are interested to store information such as unique loan\_number, Loan\_amount. Each branch maintains many loan accounts. Each customer is allowed to take many loans and at the same time a loan may be jointly taken by many Customer.



The relational schema corresponding to this ER diagram is below-

**Account**(account-number, balance, branch-name) branch-name References Branch

**Branch**(branch-name, branch-city, assets)

**Loan**(loan-number, amount, branch-name) branch-name References Branch

**Customer**(customer-name, customer-city, customer-street)

**Borrower**(customer-name,loan-number) customer-name References Customer & loan-number References Loan.

**Depositor**(account-number,customer-name) account-number References Account, customer-name References Customer

# Schema Diagrams

A database schema, along with primary key and foreign key dependencies, can be depicted by **schema diagrams**.

Each relation appears as a box, with the relation name at the top in box, and the attributes listed inside the box. **Primary key** attributes are shown **underlined**. **Foreign key** dependencies appear as **arrows from the foreign key attributes of the referencing relation to the primary key** of the referenced relation.

Schema diagram corresponding to Relational schema is given in the next slide.

# Relational Schema & Schema Diagram

**Account**(account-number, balance, branch-name) branch-name References Branch

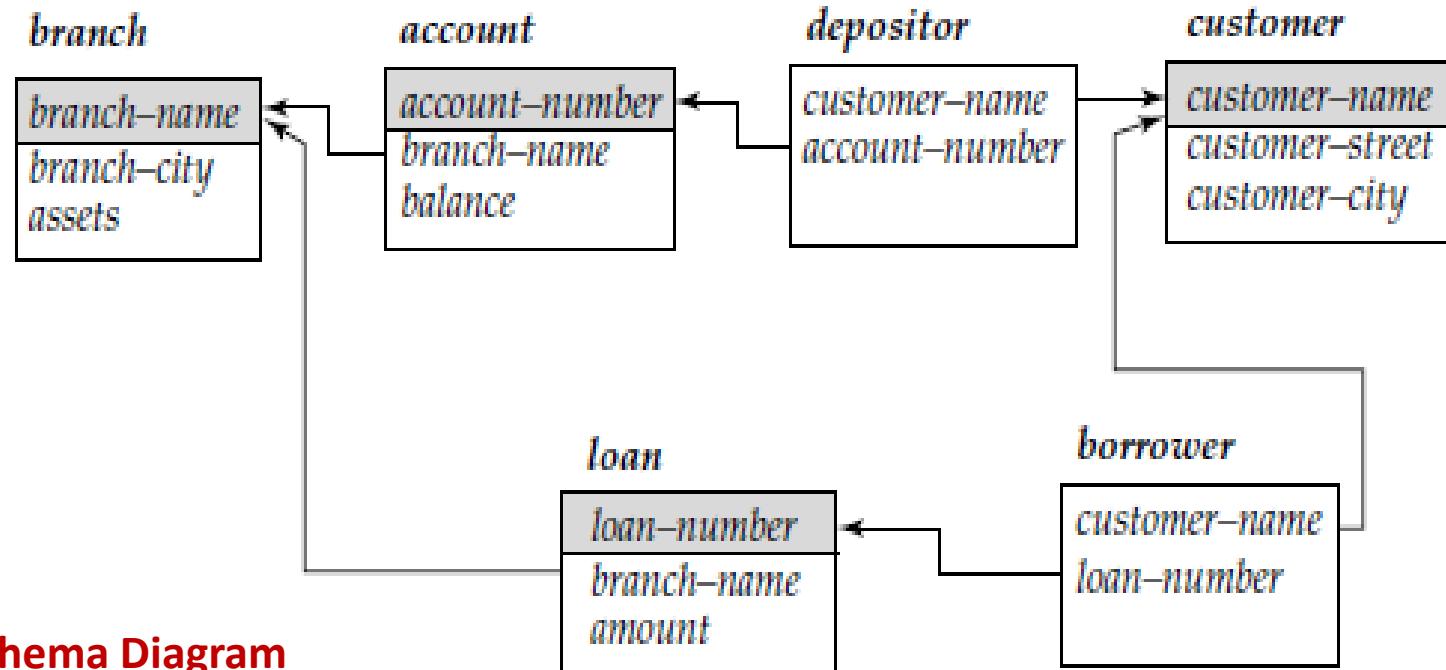
**Branch**(branch-name, branch-city, assets)

**Loan**(loan-number, amount, branch-name) branch-name References Branch

**Customer**(customer-name, customer-city, customer-street)

**Borrower**(customer-name,loan-number) customer-name References Customer & loan-number References Loan.

**Depositor**(account-number,customer-name) account-number References Account, customer-name References Customer



**Oracle SQL Commands Corresponding to Relational schema given in Previous Slide(For Lab implementation)**

**CREATE TABLE BRANCH** (BRANCH-NAME VARCHAR2(15) PRIMARY KEY, BRANCH-CITY VARCHAR2(20), ASSETS NUMBER(10,2));

**CREATE ACCOUNT** (ACCOUNT-NUMBER VARCHAR2 (3) PRIMARY KEY, BALANCE NUMBER(8,2) CHECK(BALANCE >1000), BRANCH-NAME VARCHAR2(15) NOT NULL REFERENCES BRANCH)

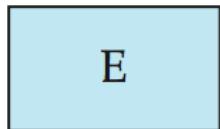
**CREATE TABLE LOAN**(LOAN-NUMBER VARCHAR2(3) PRIMARY KEY, AMOUNT NUMBER(5,1), BRANCH-NAME VARCHAR2(15) REFERENCES BRANCH)

**CREATE TABLE CUSTOMER** (CUSTOMER-NAME VARCHAR2(10) PRIMARY KEY, CUSTOMER-CITY VARCHAR2(110),CUSTOMER-STREET VARCHAR2(120))

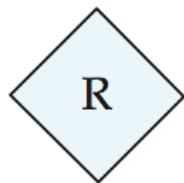
**CREATE TABLE BORROWER** (CUSTOMER-NAME VARCHAR2(10) REFERENCES CUSTOMER,LOAN-NUMBER VARCHAR2(3) REFERENCES LOAN, PRIMARY KEY(CUSTOMER-NAME, LOAN-NUMBER));

**CREATE TABLE DEPOSITOR** (ACCOUNT-NUMBER VARCHAR2 (3) REFERENCES ACCOUNT,CUSTOMER-NAME VARCHAR2(10) REFERENCES CUSTOMER), PRIMARY KEY(CUSTOMER-NAME, ACCOUNT-NUMBER));

# Summary of Symbols Used in E-R Notation



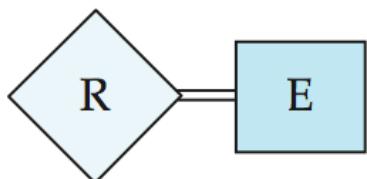
entity set



relationship set



identifying  
relationship set  
for weak entity set



total participation  
of entity set in  
relationship

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

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

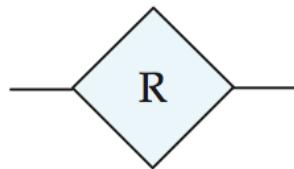
E
<u>A1</u>

primary key

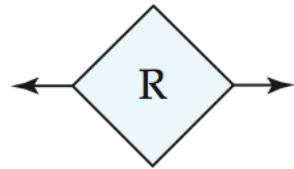
E
A1
.....

discriminating  
attribute of  
weak entity set

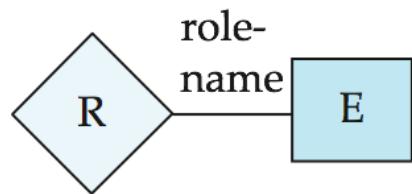
# Symbols Used in E-R Notation (Cont.)



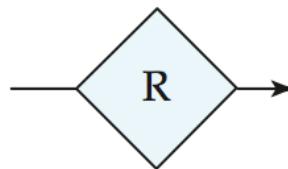
many-to-many  
relationship



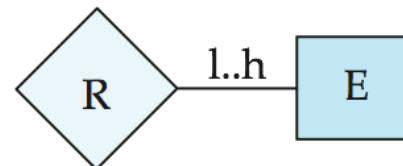
one-to-one  
relationship



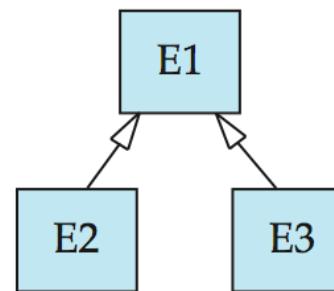
role  
indicator



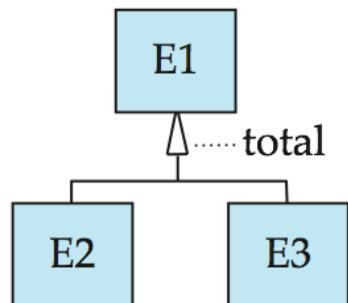
many-to-one  
relationship



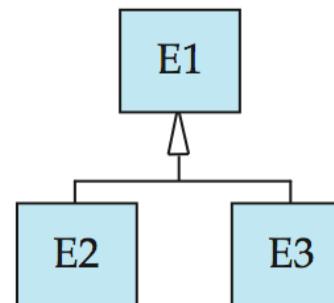
cardinality  
limits



ISA: generalization  
or specialization



total (disjoint)  
generalization

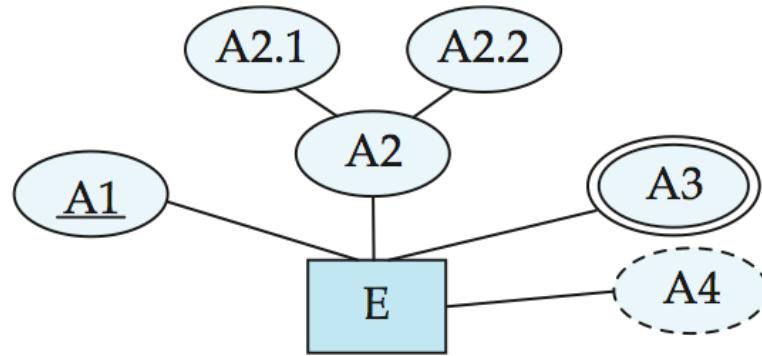


disjoint  
generalization

# Alternative ER Notations

- Chen, IDE1FX, ...

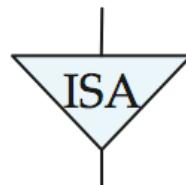
entity set E with  
simple attribute A1,  
composite attribute A2,  
multivalued attribute A3,  
derived attribute A4,  
and primary key A1



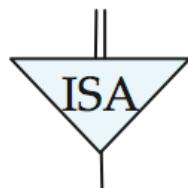
weak entity set



generalization



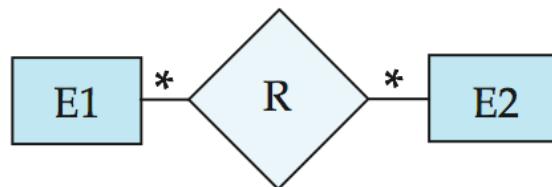
total  
generalization



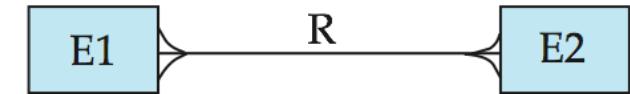
# Alternative ER Notations

Chen

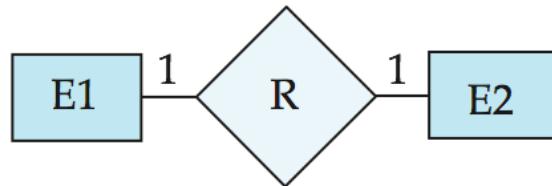
many-to-many  
relationship



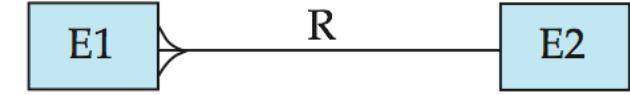
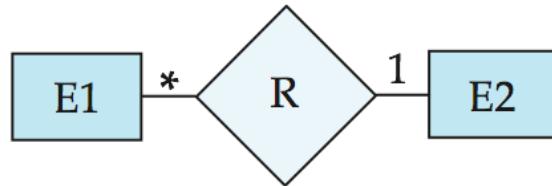
IDE1FX (Crows feet notation)



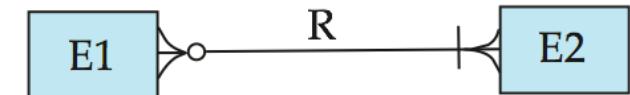
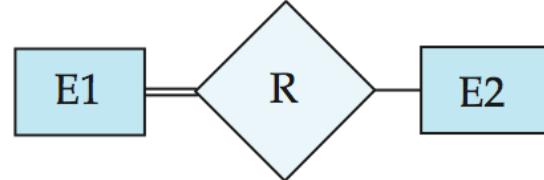
one-to-one  
relationship



many-to-one  
relationship



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



# Summary

- A **database-management system** (DBMS) consists of a collection of interrelated data.
- Database design mainly involves the design of the database schema. The **entity-relationship (E-R)** data model is a widely used data model for database design.
- An **entity** is an object that exists in the real world and is distinguishable from other objects by the properties(attributes) it is having.
- A **relationship** is an association among several entities. A **relationship set** is a collection of relationships of the same type, and an **entity set** is a collection of entities of the same type.
- The terms **super key**, **candidate key**, and **primary key** apply to entity and relationship sets as they do for relation schemas.
- **Mapping cardinalities** express the number of entities to which another entity can be associated via a relationship set.
- An entity set that does not have sufficient attributes to form a primary key is termed a **weak entity set**. An entity set that has a primary key is termed a **strong entity set**.
- **Specialization** and **generalization** define a containment relationship between a higher-level entity set and one or more lower-level entity sets.

## Some Questions

1. Explain the difference between a weak and a strong entity set.
2. Explain the distinctions among the terms primary key, candidate key, and super key with an example.
3. A weak entity set can always be made into a strong entity set by adding to its attributes the primary-key attributes of its identifying entity set. Outline what sort of redundancy will result if we also take into account of cardinality also while converting to schema.
4. Is an weak entity has to have Total Participation ?
5. Construct an E-R diagram for a car-insurance company whose customers own one or more cars each. Each car has associated with it zero to any number of recorded accidents.

ER MODEL

END OF CHAPTER

# PL/SQL

# What is PL/SQL

- Procedural Language – SQL
- An extension to SQL with design features of programming languages (procedural and object oriented)
- PL/SQL and Java are both supported as internal host languages within Oracle products.

# Why PL/SQL

- Acts as **host language** for stored procedures and triggers.
- Provides the ability to add middle tier **business logic** to client/server applications.
- Improves **performance** of multi-query transactions.
- Provides **error handling**

# PL/SQL BLOCK STRUCTURE

## **DECLARE**

(*optional*)

- variable declarations

## **BEGIN**

(*required*)

- SQL statements
- PL/SQL statements or sub-blocks

.....

## **EXCEPTION**

(*optional*)

- actions to perform when errors occur

.....

## **END;**

(*required*)

# PL/SQL Block Types

## Anonymous

```
DECLARE  
BEGIN  
    -statements  
EXCEPTION  
END;
```

a.sql

## Procedure

```
PROCEDURE <name>  
IS  
BEGIN  
    -statements  
EXCEPTION  
END;
```

p.sql

## Function

```
FUNCTION <name>  
RETURN <datatype>  
IS  
BEGIN  
    -statements  
EXCEPTION  
END;
```

f.sql

# PL/SQL Variable Types

- Scalar (char, varchar2, number, date, etc)
- Composite (%rowtype)

# DECLARE

## Syntax

```
identifier [CONSTANT] datatype [NOT NULL]  
[ := | DEFAULT expr] ;
```

## Examples

Notice that PL/SQL includes all SQL data types, and more...

### Declare

```
birthday      DATE ;  
age           NUMBER(2) NOT NULL := 27 ;  
name          VARCHAR2(13) := 'Levi' ;  
magic         CONSTANT NUMBER := 77 ;  
valid         BOOLEAN NOT NULL := TRUE ;
```

# PL/SQL- Assignment

- All variables must be declared before their use.
- The assignment statement

**:** **=**

is **not the same as** the **equality =**(comparison)  
operator

**=**

- All statements end with a ; semicolon

# DBMS\_OUTPUT.PUT\_LINE()

- **Printing on the screen**
- DBMS\_OUTPUT is the package , defined with function PUT\_LINE( string variable ) .
- string variable value passed can be displayed on the screen.
- Before using DBMS\_OUTPUT.PUT\_LINE( ..) , use SET SERVEROUTPUT ON at SQL prompt

## Example:

```
SET SERVEROUTPUT ON
```

```
DBMS_OUTPUT.PUT_LINE('HELLO ....');
```

```
DBMS_OUTPUT.PUT_LINE('MY Register Number ' ||to_char(12345));
```

|| symbol concatenates two strings.

# PL/SQL FIRST PROGRAM

```
SET SERVEROUTPUT ON
DECLARE
    message  varchar2(20):= 'Hello, World! ';
BEGIN
    dbms_output.put_line(message);
END;
/
```

Save the file as first.sql in some folder with path say d:\plsql  
Use @ d:\plsql\first.sql to run

# PL/SQL Sample Program

```
/* Find the area of the circle*/
SET SERVEROUTPUT ON
DECLARE
    pi constant number:=3.14;
    radius number:=2;
    area number;
BEGIN
    area:=pi*radius*radius;
    dbms_output.put_line('Area of circle is:'||area);
END;
/
```

# PL/SQL sample program

--Find the area of the circle

SET SERVEROUTPUT ON

DECLARE

pi constant number:=3.14;

radius number:=&radius; --prompts the user to input  
area number;

BEGIN

area:=pi\*power(radius,2);

dbms\_output.put\_line('Area of circle is:'||area);

END;

/

# Retrieving Column values into variables

Syntax:

```
SELECT Column1,Column2, . . . INTO  
Variabl1, Variabl2, . . . FROM  
table.. WHERE . . . ;
```

**Note:** This select statement must retrieve one single record

# Tables to use in PL/SQL Example

- Create table circle(radius number(2),area number(5,1), circum number(5,1))
- Insert into circle(radius) values(2);
- Insert into circle(radius) values(3);
- Insert into circle(radius) values(4);

## Comments:

Single line comments -- This is 1<sup>st</sup> PL/SQL block

Multiline comments /\* This is 1<sup>st</sup> PL/SQL block ... \*/

**SELECT.. INTO..**

**DECLARE**

v\_radius number(2);

V\_area number(5,1);

**BEGIN**

**SELECT radius INTO v\_radius FROM circle WHERE  
ROWNUM = 1;**

DBMS\_OUTPUT.PUT\_LINE(' Radius = ' || v\_radius);

V\_area:=3.142\*power(v\_radius,2);

Update circle set Area=v\_area where  
radius=v\_radius;

**END; /**

# %TYPE

**%TYPE** is used to declare a field with the same type as that of a specified table's column: Assume we have **EMP(Empno,ename,sal)**

**DECLARE**

```
v_EmpName emp.ename%TYPE;  
v_empno emp.empno%TYPE;  
v_sal emp.sal%type;
```

**BEGIN**

```
v_empno:=& v_empno;  
SELECT ename,sal INTO v_EmpName,v_sal FROM emp WHERE  
empno =v_empno;  
DBMS_OUTPUT.PUT_LINE('Name = ' || v_EmpName || ' Salary '  
|| v_sal);
```

**END;**

/

# %ROWTYPE

-- %ROWTYPE is used to declare a record with the same types as found in the specified database table, view or cursor:

```
DECLARE
    v_emp emp%ROWTYPE;
BEGIN
    v_emp.empno := 10;
    v_emp.ename := 'XXXXXXX';
END;
/
```

# %ROWTYPE

Assume we have a table DEPT(Deptno, Dname, loc)

**Set serveroutput on**

**DECLARE**

**v\_dept dept%rowtype;**

**BEGIN**

**select \* into v\_dept  
from dept where deptno=10;**

**DBMS\_OUTPUT.PUT\_LINE (v\_dept.deptno);**

**DBMS\_OUTPUT.PUT\_LINE (v\_dept.dname);**

**DBMS\_OUTPUT.PUT\_LINE (v\_dept.loc);**

**END;**

**/**

# Example:

Assume that we have a table

STUD(RegNo, Name, Mark1,Mark2,Mark3)

Write a PL/SQL block to find the marks details of a student depending on the Registration Number input by the user. Also display Total and Average marks of the student.

# COMMON IN-BUILT STRING FUNCTIONS

- CHR(asciivalue)
- ASCII(string)
- LOWER(string)
- SUBSTR(string,start,substrlen)
- LTRIM(string)
- RTRIM(string)
- LPAD(string\_to\_be\_padded, spaces\_to\_pad, |string\_to\_pad\_with|)
- RPAD(string\_to\_be\_padded, spaces\_to\_pad, |string\_to\_pad\_with|)
- REPLACE(string, searchstring, replacestring)
- UPPER(string)
- INITCAP(string)
- LENGTH(string)

# COMMON IN-BUILT NUMERIC FUNCTIONS

- ABS(value)
- ROUND(value, precision)
- MOD(value, divisor)
- SQRT(value)
- TRUNC(value,|precision|)
- LEAST(exp1, exp2...)
- GREATEST(exp1, exp2...)

# Conditional logic

## Condition:

```
If <cond>  
    Then <command>  
  
    Elsif <cond2>  
        Then <command2>  
  
    Else  
        <command3>  
  
    End if;
```

## Nested conditions:

```
If <cond>  
    Then  
        If <cond2>  
            Then  
                <command1>  
  
            End if;  
        Else <command2>  
        end if;
```

# IF-THEN-ELSIF Statements

```
...
IF rating > 7 THEN
    v_message := 'You are great';
ELSIF rating >= 5 THEN
    v_message := 'Not bad';
ELSE
    v_message := 'Pretty bad';
END IF;
...
```

# CASE.. WHEN Statement

- The CASE statement selects one sequence of statements to execute among multiple sequences.

**CASE e**

**WHEN e1 THEN r1**

**WHEN e2 THEN r2**

.....

**WHEN en THEN rn**

**[ ELSE r\_else ]**

**END CASE;**

# CASE.. WHEN- Example

**DECLARE**

grade CHAR(1); **BEGIN**

grade := & grade;

**CASE grade**

**WHEN 'A' THEN DBMS\_OUTPUT.PUT\_LINE('Excellent');**

DBMS\_OUTPUT.PUT\_LINE('A - Grade');

**WHEN 'B' THEN DBMS\_OUTPUT.PUT\_LINE('Very Good');**

DBMS\_OUTPUT.PUT\_LINE('B - Grade');

**WHEN 'C' THEN DBMS\_OUTPUT.PUT\_LINE('Good');**

DBMS\_OUTPUT.PUT\_LINE('C - Grade');

**WHEN 'D' THEN DBMS\_OUTPUT.PUT\_LINE('Fair');**

DBMS\_OUTPUT.PUT\_LINE('D - Grade');

**WHEN 'F' THEN DBMS\_OUTPUT.PUT\_LINE('Poor');**

DBMS\_OUTPUT.PUT\_LINE('F - Grade');

**ELSE DBMS\_OUTPUT.PUT\_LINE('No such grade');**

**END CASE;**

**END;**

/

# Loops: Simple Loop

```
create table number_table(  
    num NUMBER(10) );
```

```
DECLARE  
    i number_table.num%TYPE := 1;  
BEGIN  
    LOOP  
        INSERT INTO number_table VALUES(i);  
        i := i + 1;  
    EXIT WHEN i > 10;  
    END LOOP;  
END;
```

# Loops: FOR Loop

```
FOR counter IN [REVERSE] initial_value .. final_value  
LOOP
```

.....

```
sequence_of_statements;
```

.....

```
END LOOP;
```

Notice that **i** is incremented automatically

# Example-Loops: FOR Loop

```
DECLARE
  i number_table.num%TYPE;
BEGIN
  FOR i IN 1..10 LOOP
    INSERT INTO number_table VALUES(i);
  END LOOP;
END;
```

Notice that i is incremented automatically

# Example-Loops: FOR Loop ( REVERSE)

```
DECLARE
  i number_table.num%TYPE;
BEGIN
  FOR i IN REVERSE 1..10 LOOP
    INSERT INTO number_table VALUES(i);
  END LOOP;
END;
```

# Example-Loops: WHILE Loop

```
DECLARE
TEN number:=10;
i      number_table.num%TYPE:=1;
BEGIN
  WHILE i <= TEN LOOP
    INSERT INTO number_table VALUES(i);
    i := i + 1;
  END LOOP;
END;
```

# Cursors

# CURSORS

- A cursor is a private memory area.
- Set of records returned by Query are stored in Cursor.
- Data in Cursor is Active Data Set
- An Oracle Cursor = VB record set = JDBC Result Set
- **Implicit cursors** are created for every query made in Oracle
- **Explicit cursors** can be declared by a programmer within PL/SQL.

# Implicit cursor

SELECT emp\_no, emp\_name, job, salary

FROM employee

WHERE dept = 'physics' ORDER BY salary DESC

1234	A. N. Sharanu	Asst. Professor	22,000.00
1345	N. Bharath	Senior Lecturer	17,000.00
1400	M. Mala	Lab Incharge	9,000.00

- ✓ Cursor is Automatically –Opened
- ✓ All Records Retrieved.
- ✓ Cursor is Closed

# Implicit Cursor Attributes

- **SQL%ROWCOUNT** Rows returned so far (Number)
- **SQL%FOUND** One or more rows retrieved (Boolean)
- **SQL%NOT FOUND** No rows found (Boolean)
- **SQL%ISOPEN** Is the cursor open (Boolean)

# Implicit Cursor

```
SET SERVEROUTPUT ON
BEGIN
    update dept set loc='&location' where deptno=&dno;
    if SQL%found then
        DBMS_OUTPUT.PUT_LINE('Department Successfully
transferred');
    end if;
    if SQL%notfound then
        DBMS_OUTPUT.PUT_LINE('Department not existing');
    end if;
END;
/
```

# Explicit cursor

## Steps to use explicit cursor

- ☞ Declare the cursor
- ☞ Open the cursor
- ☞ Fetch data from the cursor record by record
- ☞ Close the cursor

1234	A. N. Sharanu	Asst. Professor	22,000.00
1345	N. Bharath	Senior Lecturer	17,000.00
1400	M. Mala	Lab Incharge	9,000.00

# Explicit cursor

ORACLE keep track of the "current status" of the cursor through- Cursor Attributes(system variables)

- ❖ **%NOTFOUND:** Evaluates to TRUE if the last fetch is failed i.e. no more rows are left. (single word)  
*Syntax:* cursor\_name %NOTFOUND
- ❖ **%FOUND:** Evaluates to TRUE, when last fetch succeeded  
*Syntax:* cursor\_name %FOUND
- ❖ **%ISOPEN:** Evaluates to TRUE, if the cursor is opened, otherwise evaluates to FALSE.  
*Syntax:* cursor\_name %ISOPEN
- ❖ **%ROWCOUNT:** Returns the number of rows fetched.  
*Syntax:* cursor\_name %ROWCOUNT

# Explicit Cursor Control

- Declare the cursor
- Open the cursor
- Fetch a row ← **loop**
- Test for end of cursor
- Close the cursor

**CURSOR Cur\_name IS  
SELECT... FROM... WHERE..;**

**OPEN Cur\_name;**

**FETCH Cur\_name INTO var1,var2,..**

**Cur\_name% NOTFOUND / FOUND**

**CLOSE Cur\_name;**

**Example-1:** Write a PL/SQL Block to retrieve Employee name and their salary if salary is more than 3000  
Assume the tables – EMP(Empno, Ename, Sal, Deptno)  
DEPT(Deptno, Dname, Bugdet)

# Example-1

DECLARE

cursor c\_emp is -- Cursor Declaration

select ename,sal from emp where sal>=3000;

v\_ename emp.ename%TYPE;

v\_salary emp.sal%TYPE;

BEGIN

open c\_emp; -- Open Cursor

loop

fetch c\_emp into v\_ename,v\_salary; -- Fetch Record

exit when c\_emp%notfound; -- Test End of Cursor

DBMS\_OUTPUT.PUT\_LINE(v\_ename||' draws'||v\_salary||' as salary');

end loop;

DBMS\_OUTPUT.PUT\_LINE('Number records :'||c\_emp%rowcount);

close c\_emp; -- Close cursor

END;

## Example-2

Assume we have two tables- **EMP(Empno, Ename, Sal, Deptno)** Deptno references DEPT & **DEPT(Deptno, Dname, Budget);**

Write a PL/SQL block to increase salary of employees by 5%, 10% 15% depending on the Budget of their Department. Salary increment criteria is as Below-

If Budget is  $\leq 200000$  , 5%

Budget  $>200000$  and Budget  $\leq 400000$  , 10%

Budget  $> 400000$  , 15%

## Example-2..

DECLARE

cursor c\_emp is select ename,sal,Budget from emp ,Dept  
where emp.deptno=dept.deptno;

v\_ename emp.ename%TYPE;

v\_salary emp.sal%TYPE;

v\_Budget dept.budget%Type;

updated\_sal emp.sal%TYPE;

BEGIN

open c\_emp; -- Open Cursor

loop

fetch c\_emp into v\_ename,v\_salary,v\_Budget;

exit when c\_emp%notfound;

## ..Example-2

```
IF v_Budget<=200000 THEN
    updated_sal:= v_salary+v_salary*0.05;
ELSIF v_Budget>200000 AND v_Budget<= 400000 THEN
    updated_sal:= v_salary+v_salary*0.1;
ELSE
    updated_sal:= v_salary+v_salary*0.15;
END IF;
DBMS_OUTPUT.PUT_LINE(' Name : '||v_ename);
DBMS_OUTPUT.PUT_LINE(' Old Salary : '||v_salary);
DBMS_OUTPUT.PUT_LINE(' New Salary : '||updated_sal);
DBMS_OUTPUT.PUT_LINE('-----');
end loop;
close c_emp;      -- Close cursor
END;
/
```

# Explicit Cursor- cursor for loop

Cursor for loop **simplifies the usage** of explicit cursor.

In each cursor PL/SQL following procedure is needed-

- Opening Cursor
- Fetching record and variable to hold fetched values.
- Exiting from loop
- Closing loop

In cursor for loop, no need of writing above steps explicitly.

# Explicit Cursor- cursor for loop

Write a PL/SQL block to employee name and salary information of the employees who has salary more than 3000

DECLARE

cursor c\_emp is Select Ename, Sal from Emp where  
Sal>=3000;

BEGIN

for i in c\_emp

loop

DBMS\_OUTPUT.PUT\_LINE(i.ename||' draws'||i.sal||' as  
salary');

end loop;

END;

/

# Parameterized Cursor

## Example 3:

Write a PL/SQL block to retrieve employee name, salary information of employees in the following format depending on the department number entered by the user.

Assume we have table EMP(Empno, Ename, Sal, Deptno)

## Expected Output

Enter value for par\_dept: 10

Raghu draws 10900 as salary

Ravi draws 20000 as salary

# Example 3-Parameterized Cursor

```
SET SERVEROUTPUT ON
```

```
DECLARE
```

```
CURSOR cur_emp (par_dept number) IS SELECT ename, sal FROM emp  
WHERE deptno = par_dept ORDER BY ename;
```

```
v_ename emp.ename%TYPE;
```

```
v_salary emp.salary%TYPE;
```

```
BEGIN
```

```
OPEN cur_emp (& par_dept);
```

```
LOOP
```

```
    FETCH cur_emp INTO v_ename, v_salary;
```

```
    EXIT WHEN cur_emp%NOTFOUND;
```

```
    DBMS_OUTPUT.PUT_LINE(v_ename||' draws'||v_salary||' as salary');
```

```
END LOOP;
```

```
close cur_emp ;
```

```
END;
```

```
/
```

# Parameterized Cursor

## Example 4:

Write a PL/SQL block using Cursor for loop to retrieve employee name, salary information of employees in the following format depending on the department number entered by the user.

Assume we have table EMP(Empno, Ename, Sal, Deptno)

## Expected Output

Enter value for par\_dept: 10

Raghu draws 10900 as salary

Ravi draws 20000 as salary

## Example 4

```
DECLARE
CURSOR cur_emp (par_dept number) IS SELECT ename,
sal FROM emp
          WHERE deptno = par_dept ORDER BY ename;
BEGIN
for emp_rec in cur_emp(&par_dept)
LOOP
    DBMS_OUTPUT.PUT_LINE(emp_rec.ename||'
draws'||emp_rec.sal||' as salary');
END LOOP;
END;
/
```

# END-PLSQL & CURSOR

**EXCEPTIONS, PROCEDURES,  
FUNCTIONS, PACKAGES**

# Errors

- Two types of errors can be found in a program: compilation errors and runtime errors.
- There is a special section in a PL/SQL block that handles the runtime errors.
- This section is called the *exception-handling section*, and in it, runtime errors are referred to as *exceptions*.
- The exception-handling section allows programmers to specify what actions should be taken when a specific exception occurs.

# Exception Handling

- In order to handle run time errors in the program, an exception handler must be added.
- The exception-handling section has the following structure:

**EXCEPTION**

**WHEN EXCEPTION\_NAME**

**THEN**

**ERROR-PROCESSING STATEMENTS;**

- The exception-handling section is placed after the executable section of the block.

# PL/SQL Block

**DECLARE** *(optional)*

- variable declarations

**BEGIN** *(required)*

- SQL statements
- PL/SQL statements or sub-blocks

.....

**EXCEPTION** *(optional)*

- actions to perform when errors occur

.....

**END;** *(required)*

# Predetermined Internal PL/SQL Exceptions(Built-in Exceptions)

1. **DUP\_VAL\_ON\_INDEX**: Raised when an insert or update attempts to create two rows with duplicate values in columns constrained by a unique index.
2. **LOGIN\_DENIED**: Raised when an invalid username/password was used to log onto Oracle.
3. **NO\_DATA\_FOUND**: Raised when a select statement returns zero rows.
4. **NOT\_LOGGED\_ON**: Raised when PL/SQL issues an oracle call without being logged onto Oracle.
5. **PROGRAM\_ERROR**: Raised when PL/SQL has an internal problem.

# Predetermined Internal PL/SQL Exceptions(Built-in Exceptions)

6. **TIMEOUT\_ON\_RESOURCE**: Raised when Oracle has been waiting to access a resource beyond the user-defined timeout limit.
7. **TOO\_MANY\_ROWS**: Raised when a select statement returns more than one row.
8. **VALUE\_ERROR**: Raised when the data type or data size is invalid.
9. **OTHERS**: stands for all other exceptions not explicitly named
10. **ZERO\_DIVIDE**: Raised when number is divided by zero

DECLARE

## Example:1 Named Exception

```
v_empno emp.empno%TYPE;  
v_ename emp.ename%TYPE;  
v_salary emp.salary%TYPE;
```

BEGIN

```
v_empno:=&v_empno;
```

```
SELECT ename,salary INTO v_ename,v_salary FROM emp  
WHERE empno=v_empno;
```

```
DBMS_OUTPUT.PUT_LINE(v_ename||' draws'  
||v_salary||' as salary');
```

EXCEPTION

```
WHEN NO_DATA_FOUND THEN
```

```
DBMS_OUTPUT.PUT_LINE('NO such employee found');
```

END;

/

Write a PL/SQL block to handle **NO\_DATA\_FOUND** exception raised when select .. Into statement failed to fetch record.

```
SET SERVEROUTPUT ON
```

```
DECLARE
```

```
    v_num1 number:=&v_num1;
```

```
    v_num2 number:=&v_num2;
```

```
    v_result number:=0;
```

```
BEGIN
```

```
    v_result:=v_num1/v_num2;
```

```
    DBMS_OUTPUT.PUT_LINE('result is'||v_result);
```

```
EXCEPTION
```

```
    WHEN ZERO_DIVIDE THEN
```

```
        DBMS_OUTPUT.PUT_LINE('A number cannot be  
divided by zero');
```

```
END;
```

```
/
```

## Example:2 Named Exception

Write a PL/SQL block to divide a number by another number, handle the exception raised ZERO\_DIVIDE when denominator is zero.

# User Defined Exceptions

- For example, your program asks a user to enter a value for emp\_id. This value is then assigned to the variable v\_empid that is used later in the program.
- Generally, you want a positive number for an id. By mistake, the user enters a negative number.
- However, no error has occurred because emp\_id has been defined as a number, and the user has supplied a legitimate numeric value.
- Therefore, you may want to implement your own exception to handle this situation.

# User Defined Exceptions

- This type of an exception is called a *user-defined exception* because it is defined by the programmer.
- Before the exception can be used, it must be declared.
- A user-defined exception is declared in the declarative part of a PL/SQL block as shown below:

```
DECLARE  
exception_name EXCEPTION;
```

- Once an exception has been declared, the executable statements associated with this exception are specified in the exception-handling section of the block.
- The format of the exception-handling section is the same as for built-in exceptions.

# User-Defined Exception usage

DECLARE

*exception\_name* EXCEPTION;

BEGIN

IF *CONDITION* THEN

    RAISE *exception\_name*;

ELSE

...

END IF;

EXCEPTION

    WHEN *exception\_name* THEN

        ERROR PROCESSING STATEMENTS;

END;

/

DECLARE

## Example-3 User Defined Exception

```
v_empno emp.empno%TYPE;  
v_ename emp.ename%TYPE;  
v_salary emp.salary%TYPE;  
ex_invalid_id EXCEPTION;  
  
BEGIN  
v_empno:=&v_empno;  
IF v_empno <= 0 THEN  
    RAISE ex_invalid_id;  
else  
    SELECT ename,salary INTO v_ename,v_salary FROM emp WHERE  
    empno=v_empno;  
    DBMS_OUTPUT.PUT_LINE(v_ename||' draws '||v_salary||' as  
    salary');  
end if;
```

### Example-3

Write a PL/SQL block to display employee name and salary drawn by the employee with employee number entered by the user.

Raise an exception when user enters a negative employee number and handler to handle it.

## ...Example-3

EXCEPTION

**WHEN ex\_invalid\_id THEN**

DBMS\_OUTPUT.PUT\_LINE('Emp no must be greater than zero');

**WHEN NO\_DATA\_FOUND THEN**

DBMS\_OUTPUT.PUT\_LINE('NO such employee found');

END;

/

**Example:** Write a PL/SQL block to accept account number and withdrawal amount. First check the existence of entered account number, if account number exists, deduct withdrawal amount from Balance in the Account table. If New Balance is less than 1000/- then raise an exception with an error message –**Insufficient Fund**. If entered account number do not exist, system raises **NO\_DATA\_FOUND** and handle the exception with error message – **Account Number do not exist**.

# Using OTHERS Exception

**OTHERS** exception may be used in cases when some other exception is raised apart from whatever exceptions and handlers are defined in the PL/SQL Block.

## Example:

Write a PL/SQL block to accept employee number of user and display employees information such as Name and Salary. Handle any exceptions raised using OTHERS

# Example: OTHERS exception

**DECLARE**

```
ENO EMP.EMPNO%TYPE;  
--salary emp.sal%type;  
salary number(2);
```

**BEGIN**

```
ENO:=&ENO;  
SELECT SAL INTO SALARY FROM EMP WHERE EMPNO=ENO;
```

**EXCEPTION**

**/\*When no\_data\_found then**  
**DBMS\_OUTPUT.PUT\_LINE (' employee not existing');\*/**

**WHEN OTHERS THEN**

```
DBMS_OUTPUT.PUT_LINE ('Some Error occurred ...');
```

**END;**

**/**

# Using System Defined Numbered Exception

Assume that following table is created with check constraint on supplier\_id.

```
CREATE TABLE suppliers
( supplier_id number(4),
supplier_name varchar2(50),
CONSTRAINT check_supplier_id
CHECK (supplier_id BETWEEN 100 and 9999)
);
```

Whenever **check** constraint on supplier\_id is violated **ORA -02290 exception number is raised**, we can associate an Exception name to this error number ORA - 02290 and use in a PL/SQL block to handle this exception.

# Using System Defined Numbered Exception

- First declare the exception name –  
**Supplier\_ID\_Range EXCEPTION;**
- Associate This **exception** name with the **error number**  
**Pragma EXCEPTION\_INIT(Supplier\_ID\_Range,-02290);**
- **Raise** the exception when condition is met
- Write the **exception handler** to handle the exception

## Example:

DECLARE

**Supplier\_ID\_Range1 EXCEPTION;** --user defined exception  
**pragma EXCEPTION\_INIT(Supplier\_ID\_Range1,-02290);**  
-- associate Oracle error number with user defined exception name.

BEGIN

  INSERT INTO suppliers ( supplier\_id, supplier\_name )  
  VALUES ( 1, 'IBM' );

EXCEPTION

  WHEN Supplier\_ID\_Range1 THEN  
    DBMS\_OUTPUT.PUT\_LINE(' Supplier ID entered must  
    be in the range 100 to 9999');

END;

/

# Defining Your Own Error Messages with error numbers:

## Using Procedure RAISE\_APPLICATION\_ERROR()

The procedure `RAISE_APPLICATION_ERROR` lets you issue user-defined ORA- error messages from stored subprograms.

```
raise_application_error(error_number, message);
```

where **error\_number**is a negative integer in the range -20000 .. -20999.

**Message-**is a character string up to 2048 bytes long.

An application can call **raise\_application\_error** only from an executing stored subprogram (or method). When called, **raise\_application\_error** ends the subprogram and returns a user-defined error number and message to the application.

## Example: RAISE\_APPLICATION\_ERROR()

```
DECLARE
    num_emp NUMBER;
BEGIN
    SELECT COUNT(*) INTO num_emp FROM Emp;
    IF num_emp < 1 THEN
        /* Issue your own error code (ORA-20101) with your own error message. */
        raise_application_error(-20101, 'Expecting at least 100
Employees');
    ELSE
        NULL; -- Do the rest of the processing (for the non-error case).
    END IF;
END;
/
```

# Procedures and Functions

- Oracle subprograms – includes both procedures and functions.
- Both procedures and functions:
  - Can be programmed to perform a data processing task.
  - Are **named** PL/SQL blocks, and both can be coded to take **parameters** to generalize the code.
  - Can be written with declarative, executable, and exception sections.
- Functions are typically coded to perform some type of calculation.
- Primary difference – procedures are called with PL/SQL statements while functions are called as part of an expression.

# Procedures and Functions

- **Procedures and functions:**

- Normally stored in the database within package specifications – a package is a sort of wrapper for a group of named blocks.
- Can be stored as individual database objects.
- Are parsed and compiled at the time they are stored.
- Compiled objects execute faster than nonprocedural SQL scripts because nonprocedural scripts require extra time for compilation.
- Can be invoked from most Oracle tools like SQL\*Plus, and from other programming languages like C++ and JAVA.

# Procedure and Functions

## Procedure

```
PROCEDURE <name>
IS
BEGIN
    -statements
EXCEPTION
END;
```

p.sql

## Function

```
FUNCTION <name>
RETURN <datatype>
IS
BEGIN
    -statements
EXCEPTION
END;
```

f.sql

# Procedures

- Procedures are named PL/SQL blocks.
- Created/owned by a particular schema
- Privilege to execute a specific procedure can be granted to or revoked from application users in order to control data access.
- Requires CREATE PROCEDURE (to create in your schema) or CREATE ANY PROCEDURE privilege (to create in other schemas).

# CREATE PROCEDURE Syntax

```
CREATE [OR REPLACE] PROCEDURE <procedure_name>
  (<parameter1_name> <mode> <data type>,
   <parameter2_name> <mode> <data type>, ...) {AS|IS}
  <Variable declarations>;
BEGIN
  Executable statements
[EXCEPTION
  Exception handlers]
END <optional procedure name>;
```

- Unique procedure name is required.
- OR REPLACE clause replaces if existing.
- Parameters are optional – enclosed in parentheses when used.
- AS or IS keyword is used – both work identically.
- Procedure variables are declared prior to the BEGIN keyword.
- DECLARE keyword is NOT used in named procedure.

# Compiling Procedure

- To Compile/Load a procedure use either the "@" symbol or the START SQL command to compile the file. The *<SQL filename>* parameter is the .sql file that contains the procedure to be compiled.

**SQL>@ <SQL filename>**

**SQL>start <SQL filename>**

- Filename does not need to be the same as the procedure name. The **.sql** file only contains the procedure code.
- Compiled procedure is stored in the database, not the .sql file.

# Showing Compilation Errors & Execution

- Compiled procedure is stored in the database, not the .sql file.
- If the .sql file is compiled, we get message –

**Procedure created**

Otherwise,

**Warning: Procedure created with compilation errors.**

Use SHOW ERRORS command if the procedure does not compile without errors.

**SQL> show errors;**

**OR**

**SQL> SHOW ERRORS PROCEDURE Procedure\_Name;**

Use EXECUTE to run procedure.

**SQL> EXECUTE Procedure\_Name**

# USER\_ERRORS/ALL\_ERRORS

Column	Datatype	NULL	Description
OWNER	VARCHAR2(128)	NOT NULL	Owner of the object
NAME	VARCHAR2(128)	NOT NULL	Name of the object
TYPE	VARCHAR2(12)		Type of the object: <ul style="list-style-type: none"> <li>• TYPE</li> <li>• TYPE BODY</li> <li>• VIEW</li> <li>• PROCEDURE</li> <li>• FUNCTION</li> <li>• PACKAGE</li> <li>• PACKAGE BODY</li> <li>• TRIGGER</li> <li>• JAVA SOURCE</li> <li>• JAVA CLASS</li> </ul>
SEQUENCE	NUMBER	NOT NULL	Sequence number (for ordering purposes)
LINE	NUMBER	NOT NULL	Line number at which this error occurred
POSITION	NUMBER	NOT NULL	Position in the line at which this error occurred
TEXT	VARCHAR2(4000)	NOT NULL	Text of the error
ATTRIBUTE	VARCHAR2(9)		Indicates whether the error is an error (ERROR) or a warning (WARNING)
MESSAGE_NUMBER	NUMBER		Numeric error number (without any prefix)
EDITION_NAME	VARCHAR2(128)		Name of the edition in which the object is actual

These table can be queries to retrieve require error information

# Parameters

- Both procedures and functions can take parameters.
- Values passed as parameters to a procedure as arguments in a calling statement are termed *actual parameters*.
- The parameters in a procedure declaration are called *formal parameters*.
- The values stored in *actual parameters* are values **passed to** the formal parameters – the *formal parameters* are like placeholders to store the incoming values.
- When a procedure completes, the *actual parameters* are **assigned the values** of the formal parameters.
- Assignment of values from Formal to Actual & vice versa depends on **MODE** of Parameters
- A formal parameter can have one of **three possible modes**:
  - (1) IN, (2), OUT, or (3) IN OUT.

# Defining the IN, OUT, and IN OUT Parameter Modes

- **IN** – This parameter type is passed to a procedure as a **read-only value** that cannot be changed within the procedure – this is the **default mode**.
- **OUT** – this parameter type is **write-only**, and can only appear on the left side of an assignment statement in the procedure – it is assigned an **initial value** of **NULL**.
- **IN OUT** – this parameter type **combines both IN and OUT**; a parameter of this mode is passed to a procedure, and its value can be changed within the procedure.

If a procedure raises an exception, the formal parameter values are not copied back to their corresponding actual parameters.

--Procedure Example

SET SERVEROUTPUT ON

CREATE or REPLACE PROCEDURE squareNum(**x** IN number) IS

A number;

BEGIN

A:= x \* x;

dbms\_output.put\_line('Sqaure is'||A);

END;

/

DECLARE

a number;

BEGIN

a:=&a;

squareNum(**a**);

dbms\_output.put\_line(' Number is:'||a); END;

/

Save the file as say- **proce1.sql**

Compile proce1.sql

Show errors; to know compilation errors

Main- PL/SQL block name say-  
**main\_proc1.sql** which calls the procedure  
**squareNum()**

# Parameter Constraint Restrictions

- Procedures **do not allow specifying a constraint** on the parameter data type.

```
/* Invalid constraint on parameter. */  
CREATE OR REPLACE PROCEDURE proSample  
(v_Variable NUMBER(2), ...)
```

```
/* Valid parameter. */  
CREATE OR REPLACE PROCEDURE proSample  
(v_Variable NUMBER, ...)
```

# Procedure with No Parameters

**Write a procedure to display the message- Salary is > 25000 or not for the salary of employee corresponding to the employee number passed as parameter.**

```
CREATE OR REPLACE PROCEDURE DisplaySalary IS  
temp_Salary NUMBER(10,2);
```

```
BEGIN
```

```
    SELECT Salary INTO temp_Salary FROM emp  
WHERE empno=102;
```

```
    IF temp_Salary > 25000 THEN
```

```
        DBMS_OUTPUT.PUT_LINE ('Salary is >  
than 25,000.' );
```

```
    ELSE
```

```
        DBMS_OUTPUT.PUT_LINE ('Salary is <=  
than 25,000.' );
```

```
    END IF;
```

```
END ;
```

```
/
```

# Executing *DisplaySalary* Procedure

SQL> @ch13-4.sql

*Compilation*

Procedure created.

SQL> **exec DisplaySalary**

*execution*

Salary > 25,000.

PL/SQL procedure successfully completed.

# Passing IN and OUT Parameters..

**Write a procedure to display the message- Salary is > 25000 or not for the salary of employee corresponding to the employee number passed as parameter and return the salary of the employee to calling environment.**

**CREATE OR REPLACE PROCEDURE**

**DisplaySalary2 (p\_EmployeeID**

**IN CHAR, p\_Salary OUT NUMBER) IS**

**v\_Salary NUMBER(10,2);**

**BEGIN**

**SELECT Sal INTO v\_Salary FROM Emp**

**WHERE EmpNO = p\_EmployeeID;**

**IF v\_Salary > 25000 THEN**

**DBMS\_OUTPUT.PUT\_LINE ('Salary >  
25,000.');**

**ELSE**

## ..Passing IN and OUT Parameters

```
DBMS_OUTPUT.PUT_LINE ('Salary <= 25,000.') ;
END IF;
p_Salary := v_Salary;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE ('Employee not
found.');
END DisplaySalary2;
/
```

## Example 13.6 – Calling *DisplaySalary2*

```
/* PL SQL Example */

DECLARE
    v_SalaryOutput NUMBER := 0;
BEGIN
    -- call the procedure
    DisplaySalary2(100, v_SalaryOutput);
    -- display value of salary after the call
    DBMS_OUTPUT.PUT_LINE ('Actual salary: '
        || TO_CHAR(v_SalaryOutput));
END;
/
```

## Example— Using Bind Variables

- Another approach to test a procedure. This approach uses a *bind variable* in Oracle.
- A bind variable is a variable created at the SQL\*Plus prompt that is used to reference variables in PL/SQL subprograms.
- A bind variable used in this fashion must be prefixed with a colon “:” – this syntax is required.

```
/* PL SQL Example */  
SQL> var v_SalaryOutput NUMBER;  
SQL> EXEC DisplaySalary2(7566, :v_SalaryOutput);  
Salary > 15,000.  
PL/SQL procedure successfully completed.  
SQL> PRINT v_SalaryOutput;  
  
V_SALARYOUTPUT  
-----  
16250
```

# Dropping a Procedure

- The SQL statement to drop a procedure is -
- **DROP PROCEDURE <procedureName> ;**

```
SQL> DROP PROCEDURE DisplaySalary2;  
Procedure dropped.
```

# Create Function Syntax

- Like a procedure, a function can accept multiple parameters, and the data type of the return value must be declared in the header of the function.

```
CREATE [OR REPLACE] FUNCTION
<function_name> (<parameter1_name> <mode>
<data type>,
                  <parameter2_name> <mode> <data type>,
... ) RETURN <return value data type>
{AS|IS}
      <Variable declarations>
BEGIN
    Executable Commands
    RETURN (return_value);
    .
    .
    .
[EXCEPTION
    Exception handlers]
END;
```

Syntax of the RETURN statement is: **RETURN <expression>**;

## Example– No Parameters in Function

```
CREATE OR REPLACE FUNCTION RetrieveSalary
    RETURN NUMBER
IS
    v_Salary NUMBER(10,2);
BEGIN
    SELECT Sal INTO v_Salary
    FROM Emp
    WHERE Empno = 7369;
    RETURN v_Salary;
END RetrieveSalary;
/
```

## Example – Testing *RetrieveSalary* Function

```
SQL> @ RetrieveSalary
```

Function created.

```
SQL> SELECT RetrieveSalary FROM DUAL;
```

```
RETRIEVESALARY
```

```
-----
```

```
25000
```

### **Using Bind Variable**

```
SQL> var v_SalaryOutput NUMBER;
```

```
SQL> EXEC :v_SalaryOutput := RetrieveSalary;
```

PL/SQL procedure successfully completed.

```
SQL> print v_SalaryOutput;
```

```
V_SALARYOUTPUT
```

```
-----
```

```
25000
```

# Function with Parameter

(Assume the table Employee(Empno, Firstname, MiddleName, LastName);

CREATE OR REPLACE FUNCTION FullName

(p\_EmployeeID IN

employee.EmployeeID%TYPE)

**RETURN** VARCHAR2 IS

v\_FullName VARCHAR2(100) ;

v\_FirstName employee.FirstName%TYPE;

v\_MiddleName employee.MiddleName%TYPE;

v\_LastName employee.LastName%TYPE;

BEGIN

SELECT FirstName, MiddleName, LastName INTO

v\_FirstName, v\_MiddleName, v\_LastName

FROM Employee

WHERE EmployeeID = p\_EmployeeID;

## Function with Parameter

```
v_FullName := v_LastName || ',' || v_FirstName;  
  
IF LENGTH(v_MiddleName) > 0 THEN  
    v_FullName := v_FullName || ' '  
        || SUBSTR(v_MiddleName, 1, 1) || '.';  
END IF;  
  
RETURN v_FullName;  
END FullName;  
/
```

# Testing *FullName* Function

- A simple SELECT statement executed within SQL\*Plus can return the full name for any employee identifier value as shown in PL/SQL Example 13.10.

```
/* PL SQL Example 13.10 */  
SQL> SELECT FullName(101)  
      FROM Employee  
     WHERE EmployeeID = 101;
```

FULLNAME(101)

---

Rao, Rajesh A.

# Testing *FullName* Function

```
/* PL SQL Example 13.11 */
SQL> SELECT FullName(EmployeeID)
      FROM Employee
     ORDER BY FullName(EmployeeID);
```

## **FULLNAME (EMPLOYEEID)**

---

Kumar, Ravi M.  
Rao, Rajesh A.

# Dropping a Function

- The SQL statement to drop a function is -
- **DROP FUNCTION <functionName>;**

```
SQL> DROP FUNCTION FullName;  
Function dropped.
```

PACKAGE

# PACKAGES

- A *package* is a collection of PL/SQL objects grouped together under one package name.
- Packages provide a means to collect related procedures, functions, cursors, declarations, types, and variables into a single, named database object.
- Package variables – can be referenced in any procedure, function, (other object) defined within a package.

# Package Specification and Scope

- A package consists of a **package specification** and a **package body**.
  - The *package specification*, also called the **package header**.
    - Declares global variables, cursors, exceptions, procedures, and functions that can be called or accessed by other program units.
    - A package specification must be a **uniquely named database object**.
    - Elements of a package can be declared in any order. If element “A” is referenced by another element, then element “A” must be declared before it is referenced by another element. For example, a variable referenced by a cursor must be declared before it is used by the cursor.
  - Declarations of subprograms must be forward declarations.
    - This means the declaration **only includes the subprogram name and arguments**, but does not include the actual program code.

# Create Package Specification Syntax

- Basically, a package is a named declaration section.
  - Any object that can be declared in a PL/SQL block can be declared in a package.
  - Use the **CREATE OR REPLACE PACKAGE** clause.
  - Include the specification of each named PL/SQL block header that will be public within the package.
  - Procedures, functions, cursors, and variables that are declared in the package specification are *global*.

The basic syntax for a package specification is:

```
CREATE [OR REPLACE PACKAGE [ <package name>  
{AS|IS}  
      <variable declarations>;  
      <cursor declarations>;  
      <procedure and function declarations>;  
END <package name>;
```

# Declaring Procedures and Functions within a Package

- To declare a procedure in a package – specify the procedure name, followed by the parameters and variable types:

```
PROCEDURE <procedure_name> (param1 MODE  
param1datatype, param2 MODE param2datatype, ...);
```

- To declare a function in a package, you must specify the function name, parameters and return variable type:

```
FUNCTION <function_name> (param1 MODE param1datatype,  
param2 MODE param2datatype, ...)  
RETURN <return data type>;
```

# Package Body

- Contains the code for the subprograms and other constructs, such as exceptions, declared in the package specification.
- Is optional – a package that contains only variable declarations, cursors, and the like, but no procedure or function declarations does not require a package body.
- Any subprograms declared in a package specification must be coded completely in the package body. The procedure and function specifications of the package body must match the package declarations including subprogram names, parameter names, and parameter modes.

# Create Package Body Syntax

- Use the CREATE OR REPLACE PACKAGE BODY clause to create a package body. The basic syntax is:

```
CREATE [OR REPLACE] PACKAGE BODY
<package name> AS
    <cursor declaration>
    <subprogram specifications
and code>
END <package name>;
```

# Package Specification Example:

## **Example:**

Create a package containing one function which returns square of a number and a procedure that returns cube of a number.

**CREATE OR REPLACE PACKAGE Calculate1 IS**

**FUNCTION** squrs1(Num1 IN NUMBER) return Number;

**PROCEDURE** Cubes1(Num1 IN NUMBER,Num2 OUT  
NUMBER);

**END Calculate1;**

/

Save as sql file say  
pack\_calc\_spec.sql

# Package Body Example:

```
CREATE OR REPLACE PACKAGE BODY Calculate1 IS
FUNCTION sqrs1(Num1 IN NUMBER) return Number IS
RESULT NUMBER(3);
BEGIN
    RESULT:= NUM1*NUM1;
    RETURN( RESULT);
END sqrs1;
PROCEDURE Cubes1(Num1 IN NUMBER,Num2 OUT
NUMBER) IS
BEGIN
    NUM2:= NUM1*NUM1*NUM1;
END Cubes1;
END Calculate1;
/
```

Save as sql file say  
pack\_calc\_body.sql

# Compiling Package

Compile package specification first

```
SQL> @ pack_calc_spec.sql
```

**Package created.**

Compile package body

```
SQL> @ pack_calc_body.sql
```

**Package Body created.**

# Using Package in PL/SQL Block

Set Serveroutput on;

DECLARE

M1 NUMBER(3);

RESULT NUMBER(4,1);

BEGIN

M1:=&M1;

Result:=Calculate1.squrs1 (M1); -- Function CALL

DBMS\_OUTPUT.PUT\_LINE ('SQuare is '|| Result);

Calculate1.Cubes1(M1,Result);

DBMS\_OUTPUT.PUT\_LINE ('Cube is '|| Result);

END;

/

# Example Package

Create a package containing a procedure to display full name of employee corresponding to employee number entered by the user and a function to check the existence of employee with employee number.

**CREATE OR REPLACE PACKAGE EmpFullName AS**

**PROCEDURE FindEmployee(**

emp\_ID IN employee.EmployeeID%TYPE,  
emp\_FirstName OUT employee.FirstName%TYPE,  
emp\_LastName OUT employee.LastName%TYPE ) ;  
e\_EmployeeIDNotFound **EXCEPTION**;

**FUNCTION GoodIdentifier(**

emp\_ID IN employee.EmployeeID%TYPE )  
RETURN BOOLEAN;

**END EmpFullName;**

/

# Package Body

**CREATE OR REPLACE PACKAGE BODY EmpFullName AS**

-- Procedure to find employees

**PROCEDURE FindEmployee(**

emp\_ID IN employee.EmployeeID%TYPE,  
emp\_FirstName OUT employee.FirstName%TYPE,  
emp\_LastName OUT employee.LastName%TYPE ) **AS**

**BEGIN**

SELECT FirstName, LastName  
INTO emp\_FirstName, emp\_LastName  
FROM Employee WHERE EmployeeID = emp\_ID;  
-- Check for existence of employee

IF SQL%ROWCOUNT = 0 THEN

RAISE e\_EmployeeIDNotFound;

END IF;

**Exception**

WHEN e\_EmployeeIDNotFound THEN

DBMS\_OUTPUT.PUT\_LINE('No such Employee ');

**END FindEmployee;**

# Example– Package Body

```
FUNCTION GoodIdentifier(
    emp_ID    IN employee.EmployeeID%TYPE)
RETURN BOOLEAN    IS
    v_ID_Count NUMBER;
BEGIN
    SELECT COUNT(*) INTO v_ID_Count
    FROM Employee
    WHERE EmployeeID = emp_ID;
    -- return TRUE if v_ID_COUNT is 1
    RETURN (1 = v_ID_Count);
EXCEPTION
    WHEN OTHERS THEN
        RETURN FALSE;
END GoodIdentifier;
END EmpFullName;
/
```

# Calling Package Procedure/Function

```
/* PL SQL */

DECLARE
    v_FirstName      employee.FirstName%TYPE;
    v_LastName       employee.LastName%TYPE;
    search_emp_id    employee.EmployeeID%TYPE;

BEGIN
    EmpFullName.FindEmployee (&search_emp_id,
    v_FirstName, v_LastName);

    DBMS_OUTPUT.PUT_LINE ('The employee name is:'
    || v_LastName || ', ' || v_FirstName);

EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE ('Cannot find an
employee with that ID.');

END;
/
```

# Results of Calling Package Procedure

- When the employee **identifier is valid**, the code displays the employee name as shown here.

Enter value for search\_emp\_id: 100

The employee name is: Kumar, Ravi

PL/SQL procedure successfully completed.

- When the **identifier is not valid**, the exception raised within the called procedure is propagated back to the calling procedure and is trapped by the EXCEPTION section's WHEN OTHERS clause and an appropriate message is displayed as shown here.

Enter value for search\_emp\_id: 99

Cannot find an employee with that ID.

PL/SQL procedure successfully completed.

## Cursors in Packages

```
--specification
create or replace package packg_cursor is
cursor c_emp is select
    empno,ename,sal,hiredate from emp where
    deptno=10;
    r_emp c_emp%ROWTYPE;      --row type variable
procedure p_printEmps;   --procedure
end;
/
```

# Cursors in Packages – Package body

```
create or replace package body packg_cursor is
procedure p_printEmps is
    r_emp c_emp%ROWTYPE;
begin
    open c_emp;
    loop
        fetch c_emp into r_emp;
        exit when c_emp%NOTFOUND;
        DBMS_OUTPUT.put_line(r_emp.empno);
        DBMS_OUTPUT.put_line(r_emp.ename);
        DBMS_OUTPUT.put_line(r_emp.sal || '
' || r_emp.hiredate);
    end loop;
    close c_emp;
end;      end;
```

## Executing Cursors in Packages

Either we can use a PL/SQL block to call a procedure in a package.

OR

Use EXEC command

```
SQL> EXEC packg_cursor. p_printEmps
```

7782

CLARK

2450 09-JUN-81

7839

KING

5000 17-NOV-81

7934

MILLER

1300 23-JAN-82

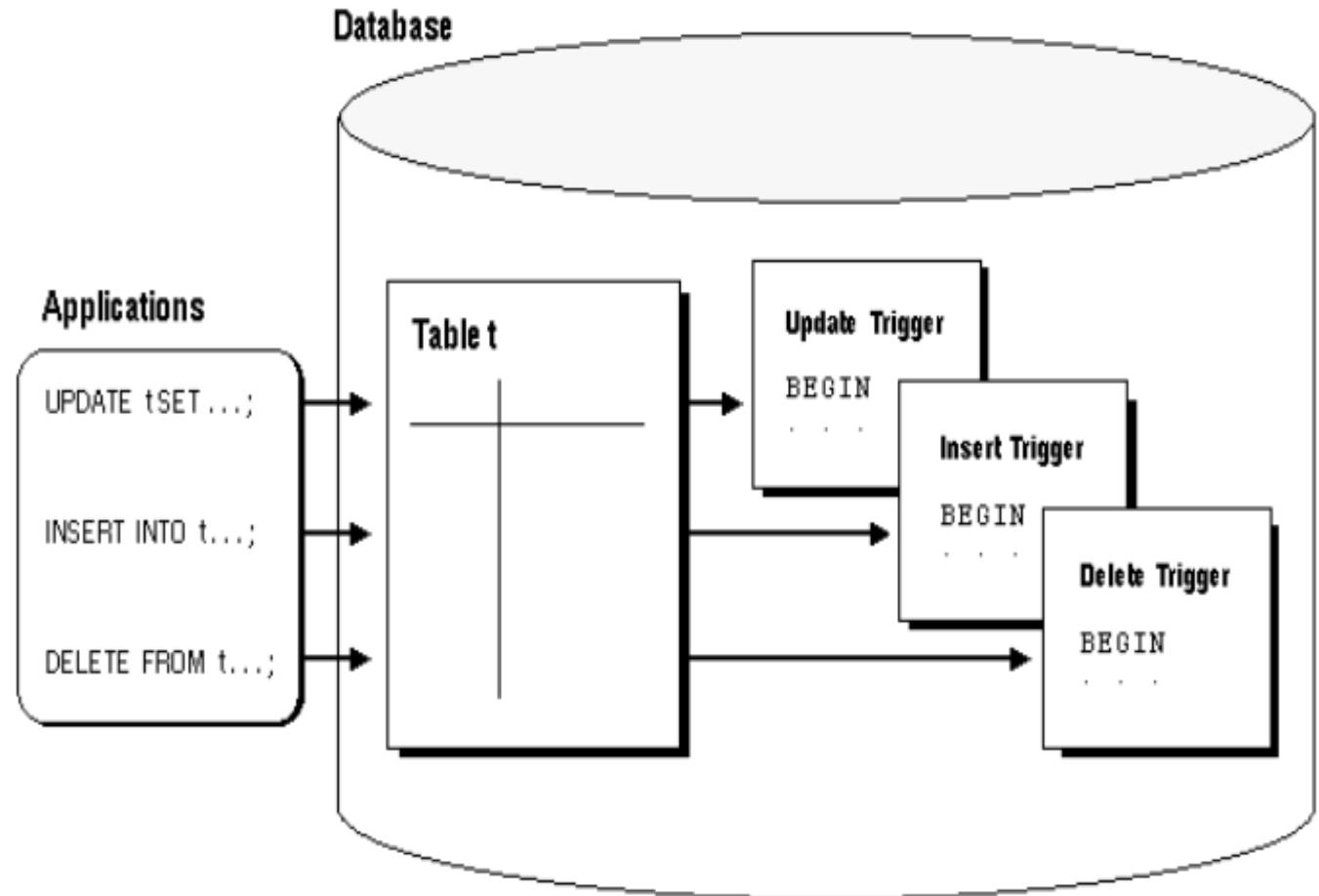
**END**

# DATABASE TRIGGERS

- **Database trigger** – a **stored** PL/SQL program unit that is **associated** with a specific database **table**, or with certain **view** types and can be fired automatically in response to any **DML events** or a **system event** such as database startup.
- Two sections:
  - A named **database event**
  - A **PL/SQL block** that will **execute** when the event occurs

# Database Trigger

Triggers get executed (fire) automatically when specified SQL DML operations – **INSERT**, **UPDATE**, or **DELETE** affecting one or more rows of a table.



# USES-DATABASE TRIGGERS

- Database triggers can be used to perform any of the following tasks:
  - Audit data modification.
  - Log events transparently.
  - Enforce complex business rules.
    - Prevent DML operations on a table after regular business hours
  - Derive column values automatically.
  - Implement complex security authorizations.
  - Maintain replicate tables.
  - Gather statistics on table access
  - Publish information about events for a publish-subscribe environment such as that associated with web programming.

# Difference between Trigger and Constraints

- Trigger affects only those rows, which are added after it is created.
- Constraints affects all the rows
  - i.e. Validates the even already existing data before defining the constraint.

## Triggers:

- are named **PL/SQL** blocks with declarative, executable, and exception handling sections.
  - are stand-alone database objects
  - do not accept arguments.
- 
- To create/test a trigger, you (not the ‘system’ user of the trigger) must have **appropriate access** to all objects referenced by a trigger action.
  - **Example:** To create a BEFORE INSERT trigger for the *employee* table requires you to have **INSERT ROW** privileges for the table.

## Create Trigger Syntax

```
CREATE [OR REPLACE] TRIGGER trigger_name  
BEFORE | AFTER | INSTEAD OF triggering_event ON  
    table_name | view_name [referencing_clause]  
[WHEN condition] [FOR EACH ROW]  
  
[DECLARE  
    Declaration statements]  
BEGIN  
    Executable statements  
[EXCEPTION  
    Exception-handling statements]  
END ;
```

The trigger body must have at least the executable section.

The declarative and exception handling sections are optional.

When there is a declarative section, the trigger body must start with the DECLARE keyword.

The WHEN clause specifies the condition under which a trigger should fire.

# Trigger Types

- **BEFORE** and **AFTER** Triggers – trigger fires before or after the triggering event.  
**Applies only to tables.**
- **INSTEAD OF** Trigger – trigger fires instead of the triggering event. **Applies only to views.**
- **Triggering event** – a DML statement issued against the table or view named in the **ON** clause – example: **INSERT, UPDATE, or DELETE.**

**DML triggers** are fired by DML statements and are referred to sometimes as **row triggers**.

- **FOR EACH ROW** clause – a **ROW trigger** that fires once **for each modified row.**
- **STATEMENT** trigger – fires **once** for the **DML statement.**
- **Referencing\_clause** – enables writing code to refer to the data in the row currently being modified by a different name.

# Conditional Predicates for Detecting Triggering DML Statement

Conditional Predicate	TRUE if and only if:
INSERTING	An INSERT statement fired the trigger.
UPDATING	An UPDATE statement fired the trigger.
UPDATING (' <i>column</i> ')	An UPDATE statement that affected the specified column fired the trigger.
DELETING	A DELETE statement fired the trigger.

```
SET SERVEROUTPUT On
CREATE OR REPLACE TRIGGER trig1
  BEFORE INSERT OR UPDATE OF sal, deptno OR
  DELETE ON emp FOR EACH ROW
BEGIN
CASE
  WHEN INSERTING THEN
    DBMS_OUTPUT.PUT_LINE('Inserting');
  WHEN UPDATING('sal') THEN
    DBMS_OUTPUT.PUT_LINE('Updating salary');
  WHEN UPDATING('Deptno') THEN
    DBMS_OUTPUT.PUT_LINE('Updating department ID');
  WHEN DELETING THEN
    DBMS_OUTPUT.PUT_LINE('Deleting');
END CASE; END;
```

**Example-1:** A trigger program to display the trigger event that resulted into trigger execution  
Save the file – **trg1\_ex.sql**

**SQL>@** trg1\_ex.sql  
Trigger created.  
If errors  
**SQL> SHOW ERRORS TRIGGER** trig1  
  
**SQL>** insert into emp(empno,  
ename, sal, deptno)  
values(119,'Akshay',3400,10);  
  
Inserting

# ROW Trigger – Accessing Rows

- Access data on the row currently being processed by using **two correlation identifiers** named **:old** and **:new**. These are special Oracle bind variables.
- The PL/SQL compiler treats the **:old** and **:new** records as records of type **trigger\_Table\_Name%ROWTYPE**.
- To reference a column in the triggering table, use the notation shown here where the *ColumnName* value is a valid column in the triggering table.

**: new . ColumnName**

**: old . ColumnName**

# Bind Variables :old and :new Defined

DML Statement	:old	:new
<b>INSERT</b>	Undefined – all column values are <b>NULL</b> as there is no “old” version of the data row being inserted.	Stores the values that will be inserted into the new row for the table.
<b>UPDATE</b>	Stores the original values for the row being updated <b>before the update</b> takes place.	Stores the new values for the row – values the row will contain <b>after the update</b> takes place.
<b>DELETE</b>	Stores the original values for the row being deleted <b>before the deletion</b> takes place.	Undefined – all column values are <b>NULL</b> as there will not be a “new” version of the row being deleted.

## Example

```
CREATE TABLE Emp (
```

```
    Empno NUMBER(4),  
    Name varchar2(10),  
    salary NUMBER(7,2),  
    Deptno  VARCHAR2(20));
```

This table contains employee information

```
CREATE TABLE Emp_log (  
    Emp_id    NUMBER(4),  
    Log_date  DATE,  
    Old_salary NUMBER(7,2),  
    New_salary NUMBER(7,2),  
    Action    VARCHAR2(20),  
    User_name varchar2(10));
```

This table records the salary change events.

**Example:** Create a trigger to store salary change information into **EMP\_log** table , when salary changes is made to **EMP** table. It has record information such as- Whose Salary has been changed, when changed, old and new values of salary, Action(**I**-increase, **D**-Decrease), User who initiated the change.

# Example-2

**CREATE OR REPLACE TRIGGER log\_salary\_increase**

**AFTER UPDATE OF sal ON emp**

**FOR EACH ROW**

**DECLARE**

    user\_action VARCHAR2(1);

**BEGIN**

    if(:new.sal>:old.sal) then

        user\_action:='I';

    elsif :new.sal=:old.sal then

        user\_action:='N';

    else

        user\_action:='D';

    end if;

    INSERT INTO Emp\_log

        VALUES (:NEW.empno, SYSDATE,:old.sal,

                 :NEW.sal,user\_action ,User);

    END;

/

**WHEN clause**- to specify condition under which Trigger has to fire

**Example 3:** Create a trigger to store salary change information into **EMP\_log** table , when salary changes above 90000 is made to **EMP** table. It has record information such as- Whose Salary has been changed, when changed, old and new values of salary, Action(I-increase, D-Decrease), User who initiated the change.

## Example-3

```
CREATE OR REPLACE TRIGGER log_salary_increase_when
```

```
AFTER UPDATE OF sal ON emp1 FOR EACH ROW WHEN
```

```
(new.sal>90000)
```

```
DECLARE
```

```
    user_action VARCHAR2(1);
```

```
BEGIN
```

```
if(:new.sal>:old.sal) then
```

```
    user_action:='I';
```

```
elsif :new.sal=:old.sal then
```

```
    user_action:='N';
```

```
else
```

```
    user_action:='D';
```

```
end if;
```

```
INSERT INTO Emp_log1
```

```
VALUES (:NEW.empno, SYSDATE,:old.sal,
```

```
:NEW.sal,user_action ,User);
```

```
END;
```

```
/
```

## STATEMENT Trigger - Example

Write a trigger to validate the salary updating action only during week-days

**Note:** Salary update statement may be updating multiple rows but Trigger is executed only once because It is **System Trigger** therefore **FOR EACH ROW** is **not used**

--User hr p: hr create another table EMP2 copy of EMP;

```
create or replace trigger sal_update_weekDays before update of sal on emp2
begin
if to_char(sysdate,'DY') = 'SUN' then
    raise_application_error(-20111,'No changes can be made on Sunday.');
end if;
end;
/
```

## STATEMENT Trigger - Example

```
/* CREATE TABLE users_log(User_name varchar2(10),Operation  
varchar2(10),Login_Date Date);*/
```

```
CREATE OR REPLACE TRIGGER note_hr_logon_trigger
```

```
AFTER LOGON ON HR.SCHEMA
```

```
BEGIN
```

```
INSERT INTO users_log VALUES (USER, 'LOGON', SYSDATE);
```

```
END;
```

```
/
```

# Using Correlation Variable- Trigger - Example

```
create table new (id number, new number);
    insert into new values(1,10);    insert into new values(2,30);
create or replace trigger trig_coRelVariable
after update on new referencing old as old_rowtyp new as new_rowtyp
for each row
begin
if updating then
    DBMS_OUTPUT.PUT_LINE('OLD VALUE'||:old_rowtyp.new||' NEW
VALUE'||:new_rowtyp.new);
else
    DBMS_OUTPUT.PUT_LINE(' SOme Error ...');
end if;
end;
/
```

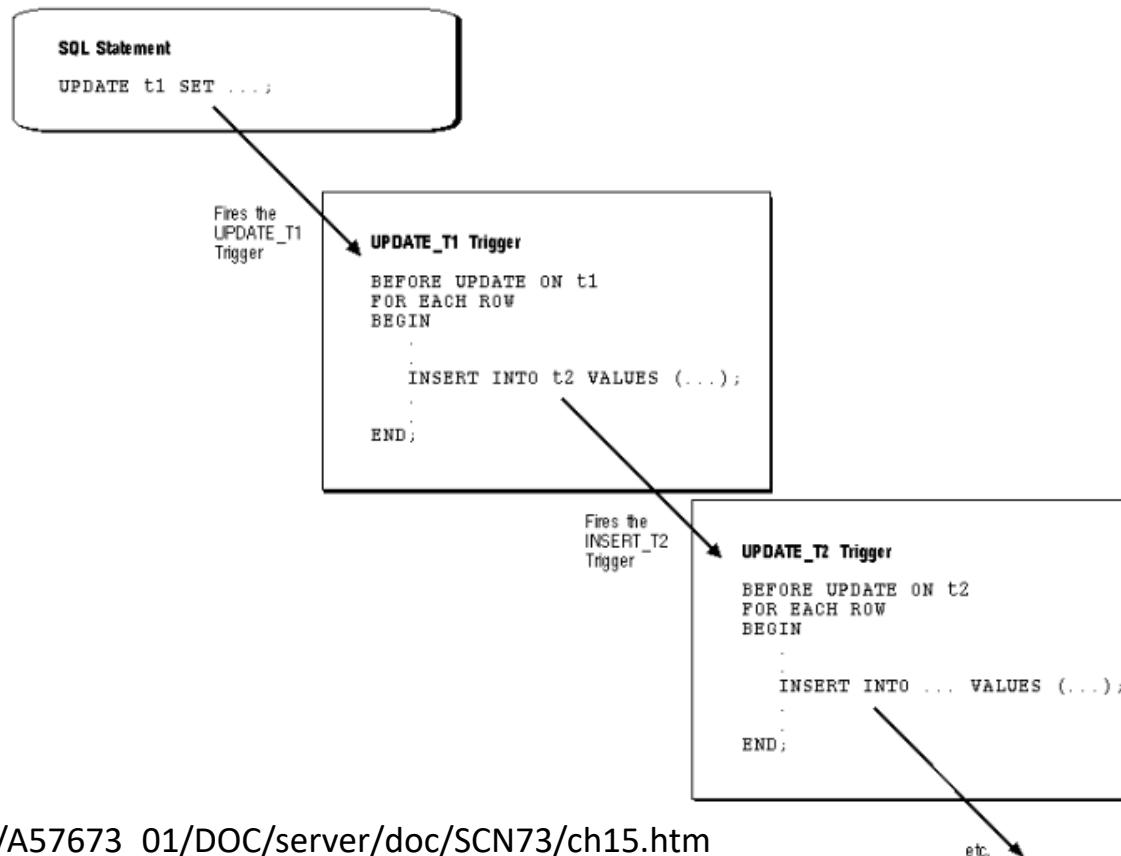
## Dropping a Trigger

- The DROP TRIGGER statement drops a trigger from the database.
- If you drop a table, all associated table triggers are also dropped.
- The syntax is:

```
DROP TRIGGER trigger_name;
```

# A Cautionary Note

When a trigger is fired, a SQL statement within its trigger action potentially can fire other triggers, . When a statement in a trigger body causes another trigger to be fired, the triggers are said to be **cascading**.



# Temporarily enabling/disabling trigger

To disable a trigger, you use the ALTER TRIGGER DISABLE statement

## Syntax:

```
ALTER TRIGGER trigger_name DISABLE;
```

### Example:

```
ALTER TRIGGER sal_update_weekDays_trg DISABLE;
```

To **disable all** triggers on a Table

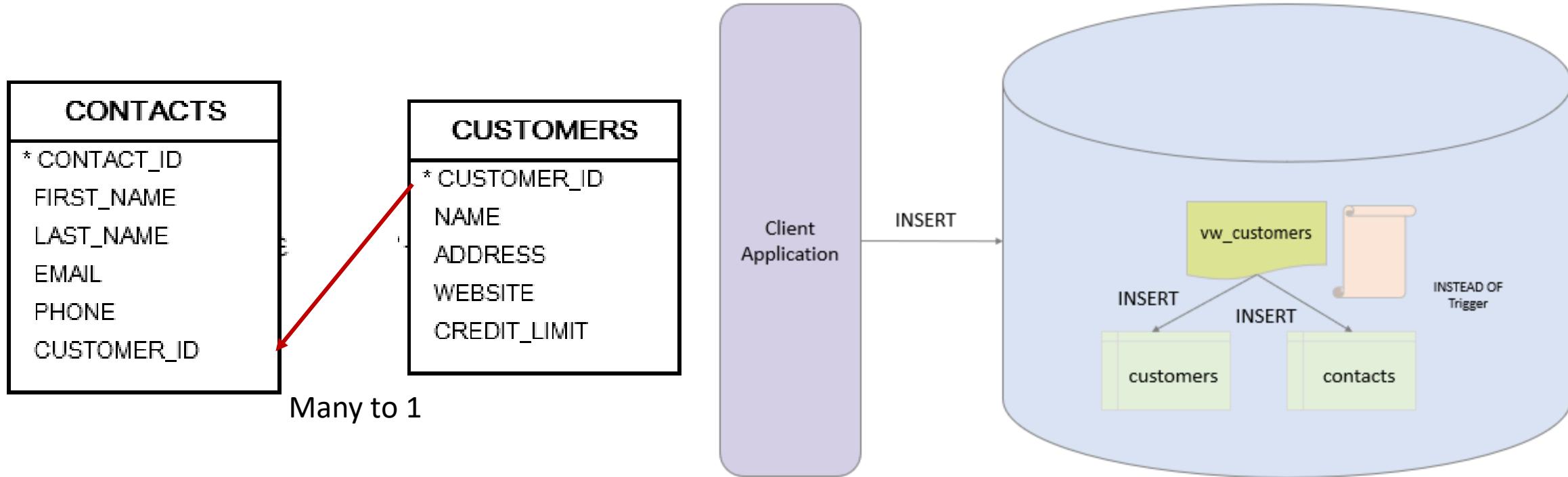
## Syntax:

```
ALTER TABLE table_name DISABLE ALL TRIGGERS;
```

### Example :

```
ALTER TABLE Emp DISABLE ALL TRIGGERS;
```

# INSTEAD OF Trigger



```
CREATE VIEW vw_customers AS SELECT name, address, website,  
credit_limit, first_name, last_name, email, phone FROM  
customers Cust, contacts Cont where  
Cust.Customer_id=Cont.Customer_id;
```

# INSTEAD OF Trigger

**Example:** Create a view VW\_emp3\_dept3 based on Emp(Empno, ename, sal, deptno) & Dept(Deptno, Dname, Loc). Write a trigger to update base tables Dept,Emp whenever VW\_emp\_dept view is updated.

```
set serveroutput on;
CREATE OR REPLACE TRIGGER emp_dept_InsteadoF_trg
  INSTEAD OF INSERT ON VW_emp3_dept3  FOR EACH ROW
DECLARE
BEGIN
  DBMS_OUTPUT.PUT_LINE(' Entered deptno '||:new.deptno);
  insert into dept3 values(:new.deptno,:new.dname,null);
  insert into emp3 (empno,ename,deptno) values(:new.empno,:new.ename,:new.deptno);
end;
/
```

END

**DATABASE MANAGEMENT SYSTEM –  
MCA  
III Semester  
JAN-2023**

# Relational Database Design

- Features of Good Relational Design
- Atomic Domains and First Normal Form
- Decomposition Using Functional Dependencies
- Functional Dependency Theory
- Algorithms for Functional Dependencies
- More Normal Form
- Database-Design Process
- Modeling Temporal Data

# Features of a Good relational database design

- Minimum **Redundancy**
  - Storage of same data in more than one location - **redundancy**
  - Disadvantage: Wastage of storage space
  - Results in updation anomalies
- Fewer **null values** in tuples
  - Values of all the attributes of a tuple are not known, so null value is to be stored
  - Cannot be provided to primary key
  - Wastage of storage space
  - Interpreting them in aggregate functions become difficult

# Update Anomalies

- **Insertion anomaly:** It leads to a situation in which certain information cannot be inserted into a relation unless some other information is stored.
- **Deletion anomaly:** It leads to a situation in which deletion of data representing certain information results in losing data representing some other information that is associated with it
- **Modification Anomaly:** It leads to a situation in which repeated data changed at one place results in inconsistency unless the same data are also changed at other places

## Example: Insert, Delete & Update Anomalies

StudentNum	CourseNum	Student Name	Address	Course
S21	9201	Jones	Edinburgh	Accounts
S21	9267	Jones	Edinburgh	Accounts
S24	9267	Smith	Glasgow	physics
S30	9201	Richards	Manchester	Computing
S30	9322	Richards	Manchester	Maths

**INSERT:** Can not insert A course if No student Opted that Course

**DELETE:** Assume S24 is the only one student who opted Physics. When S24 leaves the course then Student record is deleted which also result in loss of Course information.

**UPDATE:** If S21 changes his **Address** then in both records **Address** is to be modified otherwise it results into inconsistency.

# Combine Schemas?

- Suppose we combine *instructor* and *department* into ***inst\_dept***
  - Inst\_dept(ID,Name,Salary,Dept\_Name,Building,Budget)
  - (*Do not misunderstand as relationship set **inst\_dept***)
- Result is possible **repetition of information**

<i>ID</i>	<i>name</i>	<i>salary</i>	<i>dept_name</i>	<i>building</i>	<i>budget</i>
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

Redundancy

# A Combined Schema Without Repetition

- Consider combining relations
  - ▶ *sec\_class (sec\_id, building, room\_number)* and
  - ▶ *Section (course\_id, sec\_id, semester, year)*
- into one relation
  - ▶ *Section (course\_id, sec\_id, semester, year, building, room\_number)*
- No repetition in this case - This is a Special case. It is not TRUE always.
- Combining Relation may result into Redundancy.

# What About Smaller Schemas?

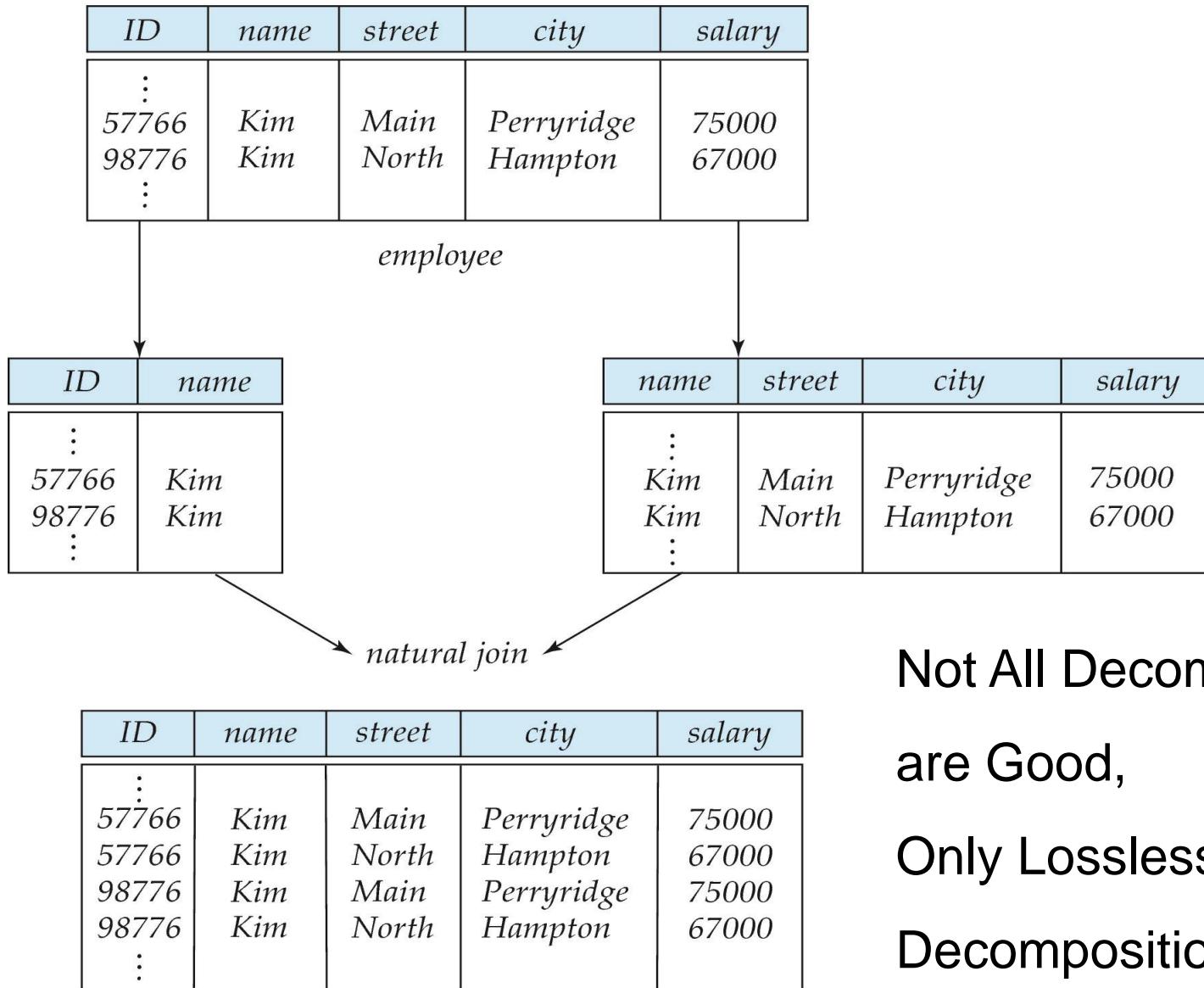
- Suppose we had started with *inst\_dept*. How would we know to split up (**decompose**) it into *instructor* and *department*?
  - **Not all decompositions are good.** Suppose we decompose

*employee*(*ID*, *name*, *street*, *city*, *salary*) into

*employee1* (*ID*, *name*) and

*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**.

# A Lossy Decomposition



Not All Decompositions  
are Good,  
Only Lossless  
Decomposition is Good

# Example of Lossless-Join Decomposition

- Lossless join decomposition
- Decomposition of  $R = (A, B, C)$   
 $R_1 = (A, B) \quad R_2 = (B, C)$

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

$r$

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

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

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

How to avoid redundancy & achieve Lossless Decomposition

# Decomposition of a Relation

- While designing a relational database certain problems that are met due to redundancy and null values can be resolved by decomposing a relation
- In decomposing, a relation is replaced with a collection of smaller relations with specific relationship between them
- Formally, the **decomposition** of a relation schema R is defined as its replacement by a set of relation schemas such that each relation schema contains a subset of attributes of R.

# 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  $R_1, R_2, \dots, R_n$  is in **good form**
  - the decomposition is a **lossless-join decomposition**
- Our theory is based on:
  - functional dependencies

# Decomposition using Functional Dependencies

- Functional dependency is a **relationship between two attributes**.
- Functional dependencies are the **special forms of integrity constraints** that **generalize the concepts of keys**
- They play a key role in developing a good database schema
- Assume that  $K$  is some attribute/s of  $R$ 
  - i.e.  $K \subseteq R$
  - we say that  $K$  is a **super key of  $r$  ( $R$ )** if the functional dependency  $K \rightarrow R$  holds on  $r$  ( $R$ ).

# 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: Functional Dependency

$R=\{A, B, C, D\}$

Let  $\alpha=\{A\}$  &  $\beta= \{B\}$

$\alpha$  &  $\beta$  are subsets of  $R$

Is  $A \rightarrow B$  holds ?

Is  $A \rightarrow C$  holds?

Let  $\alpha=\{A, B\}$  &  $\beta= \{C, D\}$

$\alpha$  &  $\beta$  are subsets of  $R$

Is  $(A, B) \rightarrow (C, D)$  holds?

Is  $(A, C) \rightarrow D$  holds ?

$A$	$B$	$C$	$D$
$a_1$	$b_1$	$c_1$	$d_1$
$a_1$	$b_2$	$c_1$	$d_2$
$a_2$	$b_2$	$c_2$	$d_2$
$a_2$	$b_3$	$c_2$	$d_3$
$a_3$	$b_3$	$c_2$	$d_4$

**Note:** that Functional dependency is not determined by the data appearing in a relation at a given point of time, instead it **depends on the meaning(semantics) of the attributes.**

# Functional Dependencies

If one set of attributes in a table determines another set of attributes in the table, then the second set of attributes is said to be functionally dependent on the first set of attributes.

## Example 1

ISBN	Title	Price
0-321-32132-1	Balloon	\$34.00
0-55-123456-9	Main Street	\$22.95
0-123-45678-0	Ulysses	\$34.00
1-22-233700-0	Visual Basic	\$25.00

Table Scheme: {ISBN, Title, Price}  
Functional Dependencies:  $\{ISBN\} \rightarrow \{Title\}$   
 $\{ISBN\} \rightarrow \{Price\}$

# Functional Dependencies

## Example 2

PubID	PubName	PubPhone
1	Big House	999-999-9999
2	Small House	123-456-7890
3	Alpha Press	111-111-1111

Table Scheme: {PubID, PubName, PubPhone}  
Functional Dependencies:  $\{PubId\} \rightarrow \{PubPhone\}$   
 $\{PubId\} \rightarrow \{PubName\}$   
 $\{PubName, PubPhone\} \rightarrow \{PubID\}$

## Example 3

AuID	AuName	AuPhone
1	Sleepy	321-321-1111
2	Snoopy	232-234-1234
3	Grumpy	665-235-6532
4	Jones	123-333-3333
5	Smith	654-223-3455
6	Joyce	666-666-6666
7	Roman	444-444-4444

Table Scheme: {AuID, AuName, AuPhone}  
Functional Dependencies:  $\{Auld\} \rightarrow \{AuPhone\}$   
 $\{Auld\} \rightarrow \{AuName\}$   
 $\{AuName, AuPhone\} \rightarrow \{AuID\}$

# Functional Dependencies

## Example 3

Assume **ITEMS(it\_name, comp\_name, price)** is a schema storing items(toothbrush, toothpaste etc.) and company(Colgate, Pepsodent etc.) names and price.

It_name	Comp_name	price
Toothpast	Colgate	20
Toothpast	Pepsodent	30
Toothbrush	Colgate	55
Toothbrush	Pepsodent	90

An It\_name may be produced by many companies ,  
therefore **It\_name  $\not\rightarrow$  Comp\_Name**  
(  $\not\rightarrow$  means NOT Functionally depends )

A company may be producing many It\_name,  
therefore **Comp\_Name  $\not\rightarrow$  It\_name**

# Functional Dependencies

## Example 3 (Contd.)

It_name	Comp_name	price
Toothpast	Colgate	20
Toothpast	Pepsodent	30
Toothbrush	Colgate	55
Toothbrush	Pepsodent	90

Similarly price is not functionally depends on It\_name or Comp\_name

$\text{It\_name} \not\rightarrow \text{Price \& Comp\_Name} \not\rightarrow \text{rice}$

However a Price is uniquely determined by the combination of  
(It\_Name,Comp\_Name), therefore

$(\text{It\_Name}, \text{Comp\_Name}) \rightarrow \text{Price}$

# Trivial Functional Dependency

- A functional dependency is **trivial** if it is satisfied by all relations
  - $X \rightarrow X$  is satisfied by all relations having attribute X.
  - Similarly,  $XY \rightarrow X$  is satisfied by all relations having attributes X and Y.
- Example:
  - ▶  $ID, name \rightarrow ID$
  - ▶  $name \rightarrow name$

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

- The dependencies that are not trivial are called **non-trivial dependencies**.
- These are the dependencies that correspond to the essential integrity constraints.

# Functional Dependencies

## Super Key & Candidate key

- $K$  is a **super key** for relation schema  $R$  if and only if  $K \rightarrow R$
- $K$  is a **candidate key** (minimal Super Key) for  $R$  if and only if
  - $K \rightarrow R$  (*means  $K$  is Super key*), and
  - for **no**  $\alpha \subset K$ ,  $\alpha \rightarrow R$

# Example: Minimal Super Key

□ Consider the ITEMS relation.

Let us take  $K = (\text{It\_Name}, \text{Comp\_Name})$

A Price is uniquely determined by the combination of  
(It\_Name, Comp\_Name), therefore

$(\text{It\_Name}, \text{Comp\_Name}) \rightarrow \text{Price}$

Following two functional dependency also holds because of trivial functional dependency  
( i.e.  $\alpha \rightarrow \beta$  holds always if  $\beta \subset \alpha$ )

$(\text{It\_Name}, \text{Comp\_Name}) \rightarrow \text{It\_Name}$  Trivial

because It\_Name  $\subset (\text{It\_Name}, \text{Comp\_Name})$

$(\text{It\_Name}, \text{Comp\_Name}) \rightarrow \text{Comp\_Name}$

because Comp\_Name  $\subset (\text{It\_Name}, \text{Comp\_Name})$

Therefore  $(\text{It\_Name}, \text{Comp\_Name}) \rightarrow \text{ITEMS}$

$(\text{It\_Name}, \text{Comp\_Name})$  is Super Key for ITEMS

Is it Minimal Super key also?

## Example: Minimal Super Key (Contd.)

As Discussed in previous slide      **(It\_name, Comp\_name) is Super Key.**

- Further K i.e. **(It\_name, Comp\_name)** can be a **candidate key** (means minimal Super key) ,  
if **none of the subset of K determine R** i.e. ITEMS.

**“None of the sub sets of (It\_name, Comp\_name) is Super key”**

Subsets of **(It\_name, Comp\_name)** are

**(It\_name) & (Comp\_Name)**

We know **It\_name** alone do not determine **ITEMS** & **Comp\_Name** alone also do not determine **ITEMS** .

i.e. Neither **It\_name** nor **Comp\_Name** is Super Key for **ITEMS**

- i.e      **It\_name**  $\not\rightarrow$  **ITEMS** & **Comp\_name**  $\not\rightarrow$  **ITEMS**

## Example: Minimal Super Key (Contd.)

- From Super key, **(It\_name, Comp\_name)** we are unable to find any **subset** which **determine ITEMS** .

not possible to find any sub set of

**(It\_name, Comp\_name) which is super key**

- i.e. Hence **(It\_name, Comp\_name)** are **minimum attributes required** to determine ITEMS.
- Therefore **(It\_name, Comp\_name)** is minimal Super key (i.e. candidate key)

# Write the Functional Dependencies for a given System Requirements

- Assume that we want to store information about employees and department in which they are working.
- Every employee has a unique **EmpNo**. About each employee we are interested to store his **name** ,**salary** what he earns, **city** in which he resides , **pincode** , **STDCode** of the city and **PhoneNumber** of the employee. Every city has one pincode (ignoring the point that-with in a city different areas will have different Pincode) , similar type assumption we considered here like every city has one STDCode.
- Each employee works exactly in one department. More than one employees work in a department. About each department we want to record department number-**Deptno** and department name **Dname**, Department location –**Loc**. Every department has a unique deptno, every department has one unique Dname and each department is situated in one Location.

See sample data corresponding to these constraints in next slide

## Sample data corresponding to the constraints in previous slide

Empno	Ename	Sal	City	PinCode	STD_Code	Phone_Number	Deptno	Dname	Loc
157	Niraj	1000	MANIPAL	576104	820	12345	D1	ADMINSTION	MANGALORE
100	Ravi	2000	MANGALORE	587693	819	56788	D2	MARKETING	BOMBAY
102	Raviraj	3899	BANGLAORE	580001	80	786536	D3	RESEARCH	BOMBAY
111	Raghу	5788	MANIPAL	576104	820	125978	D1	ADMINSTION	MANGALORE
150	X	2467	BOMBAY	489921	420	678899	D2	MARKETING	BOMBAY
103	Y	9763	HYDERBAD	765690	560	567899	D3	RESEARCH	BOMBAY
109	Z	46789	CHENNAI	674321	730	656890	D1	ADMINSTION	MANGALORE
125	Manu	5788	BANGLAORE	580001	80	123456	D3	RESEARCH	BOMBAY
104	A	8653	MANIPAL	576104	820	89865	D2	MARKETING	BOMBAY
106	B	79076	CHENNAI	674321	730	566778	D3	RESEARCH	BOMBAY
123	Mahesh	2000	MANGALORE	587693	819	55677	D2	MARKETING	BOMBAY
108	Ravi	44466	CHENNAI	674321	730	57780	D4	PRODUCTION	MANGALORE
124	D	44467	HYDERBAD	765690	560	786535	D5	PURCHASE	CHENNAI
113	Z	65576	MANIPAL	576104	820	89627	D5	PURCHASE	CHENNAI

# Set of Functional Dependencies that hold

Following are the some FDs that hold on the system as per the requirements given in slide 26

$\text{Empno} \rightarrow \text{Name}$ ;  $\text{Empno} \rightarrow \text{Sal}$ ;  $\text{Empno} \rightarrow \text{Deptno}$ ;  
 $\text{Empno} \rightarrow \text{Dname}$ ;  $\text{Empno} \rightarrow \text{City}$ ;  $\text{Empno} \rightarrow \text{PinCode}$ ;  
 $\text{Empno} \rightarrow \text{STD\_Code}$ ;  $\text{Empno} \rightarrow \text{Phone\_Number}$ ;  $\text{Empno} \rightarrow \text{Loc}$   
 $\text{Deptno} \rightarrow \text{Dname}$ ;  $\text{Deptno} \rightarrow \text{Loc}$ ;  $\text{Dname} \rightarrow \text{Deptno}$ ;  
 $\text{Dname} \rightarrow \text{Loc}$ ;  $\text{City} \rightarrow \text{PinCode}$ ;  $\text{PinCode} \rightarrow \text{City}$ ;  
 $\text{City} \rightarrow \text{STD\_Code}$ ;  $\text{STD\_Code} \rightarrow \text{City}$

**Whether these Functional Dependency Hold ?**

$\text{Deptno} \rightarrow \text{Ename}$ ;  $\text{Deptno} \rightarrow \text{Sal}$ ;  $\text{Dname} \rightarrow \text{Sal}$ ;  
 $\text{City} \rightarrow \text{Dname}$ ;  $\text{City} \rightarrow \text{Deptno}$ ;  $\text{Phone\_Number} \rightarrow \text{STD\_Code}$ ;  
 $\text{City} \rightarrow \text{PhoneNumber}$ ;  $\text{PinCode} \rightarrow \text{STD\_Code}$

**IS ( Empno , Name)  $\rightarrow$  Deptno ;**

**\*Closure of FD**

# Functional Dependencies(Cont'd)

## Full Functional dependency:

- If A and B are attributes of a table, B is fully functionally dependent on A if B is functionally dependent on A, but **not on any proper subset of A**.

### □ Example

Consider the example **ITEMS(It\_Name, Comp\_Name, Price)**

- Consider Functional Dependency **(It\_Name, Comp\_Name) → Price**
- **Price is Fully Functionally dependent** because-

- It\_Name  $\not\rightarrow$  Price
  - Comp\_Name  $\not\rightarrow$  Price
- } Price do not Functionally dependent on  
none of the proper subset of  
(It\_Name, Comp\_Name)

# Functional Dependencies(Cont'd)

## Partial Functional Dependency:

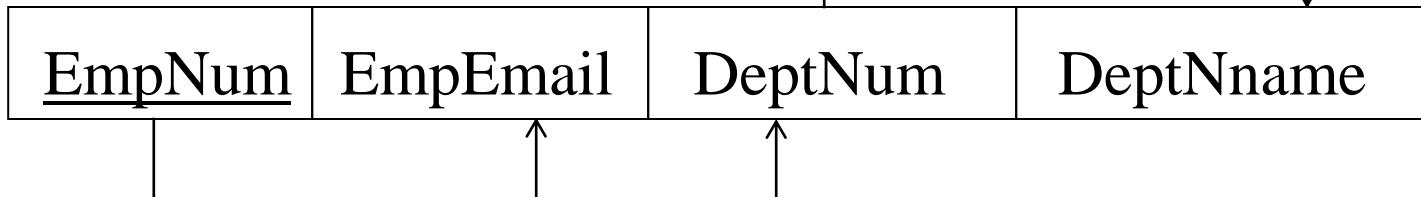
- If A and B are attributes of a table, B is partially dependent on A if there is some attribute that can be removed from A and yet the dependency still holds.
- Example

$(\text{Empno}, \text{ Name}) \rightarrow \text{Deptno}$  , but       $\text{Empno} \rightarrow \text{Deptno}$

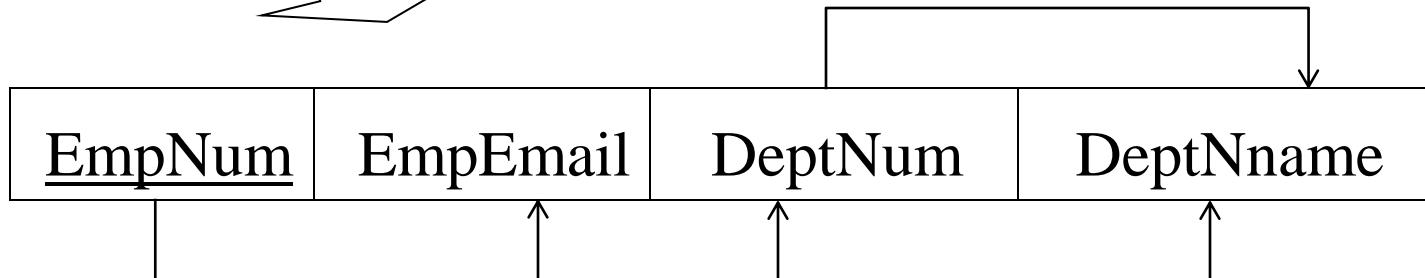
- Deptno is also dependent on Empno which is subset of (Empno,Name)
- Therefore **Deptno** is **not fully dependent** on **(Empno,Name)**,
- Means **Deptno** is **Partially Dependent** on **(Empno, Name)**

# Transitive dependency

$\text{EmpNum} \rightarrow \text{DeptNum}$



$\text{DeptNum} \rightarrow \text{DeptName}$



DeptName is *transitively dependent* on EmpNum via DeptNum

$\text{EmpNum} \rightarrow \text{DeptName}$

# Normalization

- Normalization is the process of modifying a relation schema based on its Functional Dependencies and primary keys so that it **meets certain rules** called **normal forms**.
- It is conducted by **evaluating a relation schema** to check whether it satisfies particular rules and if not, then **decomposing** the schema into set of smaller relation schemas that do not violate those rules
- Normalization can be considered as fundamental to the modelling and design of a relational database
- Main purpose is to **eliminate data redundancy** and **avoid data update anomalies**

# Normalization

- A relation is said to be in a particular normal form if it satisfies certain specified constraints
- Each of the normal form is stricter than its predecessors
- The normal forms are used to ensure that **various types of anomalies and inconsistencies are removed** from the database

# Normalization

- A properly normalized database should have the following characteristics
  - **Atomic values** in each fields
  - **Absence of redundancy.**
  - Minimal use of **null** values.
  - Minimal **loss of information.**

# Levels of Normalization

- Levels of normalization based on the amount of redundancy in the database.
  - Various levels of normalization are:
    - First Normal Form (1NF)
    - Second Normal Form (2NF)
    - Third Normal Form (3NF)
    - Boyce-Codd Normal Form (BCNF)
    - Fourth Normal Form (4NF)
    - Fifth Normal Form (5NF)
    - Domain Key Normal Form (DKNF)
- 
- The diagram consists of three vertical red arrows pointing upwards. The leftmost arrow is labeled 'Redundancy' and has a downward-pointing arrowhead at its top. The middle arrow is labeled 'Number of Tables'. The rightmost arrow is labeled 'Complexity'.

Most databases should be 3NF or BCNF in order to avoid the database anomalies.

# First Normal Form

- A relational schema R is in **first normal form** if the domains of all attributes of R are **atomic**
- 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-**Dept** & **RollNo**
- **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

# 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.

# Example

**TABLE\_PRODUCT**

Product ID	Color	Price
1	red, green	15.99
2	yellow	23.99
3	green	17.50
4	yellow, blue	9.99
5	red	29.99

# 1NF Decomposition

1. Place all items that appear in the repeating group in a new table
2. Designate a primary key for each new table produced.
3. Duplicate in the new table the primary key of the table from which the repeating group was extracted or vice versa.

# 1NF Decomposition

**TABLE\_PRODUCT\_PRICE**

Product ID	Price
1	15.99
2	23.99
3	17.50
4	9.99
5	29.99

**TABLE\_PRODUCT\_COLOR**

Product ID	Color
1	red
1	green
2	yellow
3	green
4	yellow
4	blue
5	red

# 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  $F$ .       $F^+ \supset F$

Set of Functional Dependencies that hold
Following are the some FDs that hold on the system as per the requirements given in <a href="#">slide 26</a>
$\text{Empno} \rightarrow \text{Name}$ ; $\text{Empno} \rightarrow \text{Sal}$ ; $\text{Empno} \rightarrow \text{Deptno}$ ;
$\text{Empno} \rightarrow \text{Dname}$ ; $\text{Empno} \rightarrow \text{City}$ ; $\text{Empno} \rightarrow \text{PinCode}$ ;
$\text{Empno} \rightarrow \text{STD\_Code}$ ; $\text{Empno} \rightarrow \text{Phone\_Number}$ ; $\text{Empno} \rightarrow \text{Loc}$
$\text{Deptno} \rightarrow \text{Dname}$ ; $\text{Deptno} \rightarrow \text{Loc}$ ; $\text{Dname} \rightarrow \text{Deptno}$ ;
$\text{Dname} \rightarrow \text{Loc}$ ; $\text{City} \rightarrow \text{PinCode}$ ; $\text{PinCode} \rightarrow \text{City}$ ;
$\text{City} \rightarrow \text{STD\_Code}$ ; $\text{STD\_Code} \rightarrow \text{City}$
Whether these Functional Dependency Hold ?
$\text{Deptno} \rightarrow \text{Ename}$ ; $\text{Deptno} \rightarrow \text{Sal}$ ; $\text{Dname} \rightarrow \text{Sal}$ ;
$\text{City} \rightarrow \text{Dname}$ ; $\text{City} \rightarrow \text{Deptno}$ ; $\text{Phone\_Number} \rightarrow \text{STD\_Code}$ ;
$\text{City} \rightarrow \text{PhoneNumber}$ ; $\text{PinCode} \rightarrow \text{STD\_Code}$
IS ( $\text{Empno}, \text{Name}$ ) $\rightarrow \text{Deptno}$ ;
*Closure of FD

# 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  $\alpha \rightarrow \beta$  **(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).

# Closure of Functional Dependencies (Cont.)

- 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  $\alpha\gamma \rightarrow \delta$  holds (**pseudotransitivity**)

The above rules can be inferred from Armstrong's axioms.

\* For Proof, see notes section.

## Example: Finding some logically implied functional dependencies using Armstrong Axioms

□  $R = (A, B, C, G, H, I)$

$F = \{ A \rightarrow B$

$A \rightarrow C$

$CG \rightarrow H$

$CG \rightarrow I$

$B \rightarrow H \}$

if  $\beta \subseteq \alpha$ , then  $\alpha \rightarrow \beta$

(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)

□ some members of  $F^+$

□  $A \rightarrow H$

▶ by transitivity from  $A \rightarrow B$  and  $B \rightarrow H$

□  $AG \rightarrow I$

▶ Since  $A \rightarrow C$  and  $CG \rightarrow I$  (pseudotransitivity)

□  $CG \rightarrow HI$

▶ Since  $CG \rightarrow H$  and  $CG \rightarrow I$  (Union rule)

# Procedure for Computing $F^+$

- 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 any further

# Closure of Attribute Sets

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

Algorithm to compute  $a^+$ , the **closure** of  $a$  under  $F$

```
result := a;  
  
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 for Attribute Set Closure

- $R = (A, B, C, G, H, I)$
- $F = \{ A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H \}$  Find  $AG^+$

- $(AG)^+$

1.  $result = AG$
2.  $result = ABCG$  ( $A \rightarrow C$  and  $A \rightarrow B$ )
3.  $result = ABCGH$  ( $CG \rightarrow H$  and  $CG \subseteq AGBC$ )
4.  $result = ABCGHI$  ( $CG \rightarrow I$  and  $CG \subseteq AGBCH$ )

Algorithm to compute  $a^+$ , the closure of  $a$  under  $F$

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

# Step-by-step approach to Find AG<sup>+</sup>

1.  $result = AG$
2.  $While\ loop$       ( $1^{st}$  While loop)
3.  $For\ Loop$

1. take  $A \rightarrow B$

$$A \subseteq result$$

$$result = result \cup B = \{AGB\}$$

2. take  $A \rightarrow C$

$$A \subseteq result$$

$$result = result \cup C = \{AGBC\}$$

3. Take  $CG \rightarrow H$

$$CG \subseteq result$$

$$result = result \cup H = \{AGBCH\}$$

4. Take  $CG \rightarrow I$

$$CG \subseteq result$$

$$result = result \cup I = \{AGBCHI\}$$

5. Take  $B \rightarrow H$

$$B \subseteq result$$

$$result = result \cup H = \{AGBCHI\}$$

*End of For Loop*

## ....Step-by-step approach to Find AG<sup>+</sup>

At the end of 1<sup>st</sup> While loop result= { AGBCHI}

So when 2<sup>nd</sup> While loop starts result= { AGBCHI}

1. While loop (2<sup>nd</sup> While loop)

2. For Loop

1. take **A** → **B**

$A \subseteq$  result

$result = result \cup B = \{AGBCHI\}$

2. take **A** → **C**

$A \subseteq$  result

$result = result \cup C = \{AGBCHI\}$

3. Take **CG** → **H**

$CG \subseteq$  result

$result = result \cup H = \{AGBCHI\}$

4. Take **CG** → **I**

$CG \subseteq$  result

$result = result \cup I = \{AGBCHI\}$

5. Take **B** → **H**

$B \subseteq$  result

$result = result \cup H = \{AGBCHI\}$

End of For Loop

**Result from While loop 1 and While loop 2 are same , there is no change in result , therefore exit While loop**

# Exercises

1.  $R(A, B, C, D)$

$$F = \{ A \rightarrow B, B \rightarrow C, B \rightarrow D \}, \text{ Find } A^+, B^+$$

2.  $R(A,B,C,D)$

$$F = \{ C \rightarrow D, A \rightarrow B, B \rightarrow C \} \text{ Find } A^+, C^+$$

1.

$$A^+ = \{ A, B, C, D \}$$

$$B^+ = \{ B, C, D \}$$

2.

$$A^+ = \{ A, B, C, D \}$$

$$C^+ = \{ C, D \}$$

# Uses of Attribute Closure

There are several uses of the attribute closure algorithm:

## 1. Testing for superkey:

- To test if  $\alpha$  is a superkey, we compute  $\alpha^+$ , and check if  $\alpha^+$  contains all attributes of  $R$ . i.e  $\alpha \rightarrow R$

## 2. Testing functional dependencies

- To **check** if a functional dependency  $\alpha \rightarrow \beta$  holds (or, in other words, is it 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

## 3. Computing closure of $F$ i.e. $F^+$

- For each  $\gamma \subseteq R$ , we find the closure  $\gamma^+$ , and for each  $S \subseteq \gamma^+$ , we output a functional dependency  $\gamma \rightarrow S$ .

# 1. Example Testing for superkey & Candidate Key:

□  $R = (A, B, C, G, H, I)$

□  $F = \{ A \rightarrow B$   
 $A \rightarrow C$   
 $CG \rightarrow H$   
 $CG \rightarrow I$   
 $B \rightarrow H \}$

□ Is **AG** a Super Key ? \*

□ Is **AG** a candidate key? \*

1. Is AG a super key?

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

    2. Is any subset of AG a super key?

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

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

□ Is **A** is super key?

\* See [slide 21](#) for testing – is the super key is also candidate key ?

# Exercise

- $R = (A, B, C, G, H, I)$
- $F = \{ A \rightarrow B$   
 $A \rightarrow C$   
 $CG \rightarrow H$   
 $CG \rightarrow I$   
 $B \rightarrow H$   
 $C \rightarrow G$
- }
- Is **AB** a Super Key ? \*
- Is **AB** a candidate key? \*

## Step-by-step approach to Check AG is Super Key

Is **AG** is Super Key?

- Find **AG<sup>+</sup>**
- From slide 50 we know **AG<sup>+</sup>** is **AG<sup>+</sup> = {ABCDEFGHI}**
- i.e.  $R \subseteq (AG)^+$ 
  - Means  $R = \{A, B, C, G, H, I\}$ , all attributes of R Contained in **AG<sup>+</sup>**
- Therefore **AG → R**
- Therefore **AG is Super key**

*Reference: slide 21*

- $K$  is a **super key** for relation schema  $R$  if and only if  $K \rightarrow R$
- $K$  is a **candidate key (minimal Super Key)** for  $R$  if and only if
  - $K \rightarrow R$  (*means K is Super key*), and
  - for no  $\alpha \subset K$ ,  $\alpha \rightarrow R$

# Step-by-step approach to Check AG is Candidate Key

## IS AG Candidate key ?

We know AG is Super key.

Check further, is there any subset of AG is Super Key?

Subsets of AG are A & G

Check is  $A \rightarrow R$  or  $G \rightarrow R$  ?

If any one subset is Super Key then AG is not Candidate key, but only super key.

If none of subsets are Super key then AG is Candidate Key.

Find  $A^+$  &  $G^+$      $A^+ = \{ A, B, C, H \}$   $A \not\rightarrow R$  , Therefore A is Not Super Key

$G^+ = \{ G \}$   $G \not\rightarrow R$  , Therefore G is Not Super Key

None of subsets of AG are Super Keys –

Therefore AG is Candidate Key (Minimal Super Key)

Reference: slide 21

- K is a super key for relation schema R if and only if  $K \rightarrow R$
- K is a candidate key (minimal Super Key) for R if and only if
  - $K \rightarrow R$  (means K is Super key), and
  - for no  $\alpha \subset K$ ,  $\alpha \rightarrow R$

## 2.Example for Testing functional dependencies

To check if a functional dependency  $\alpha \rightarrow \beta$  holds (or, in other words, is it in  $F^+$ ), just check if  $\beta \subseteq \alpha^+$ . If  $\beta \not\subseteq \alpha^+$  then  $\alpha \not\rightarrow \beta$

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

$$F = \{ A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H \}$$

Is  $CG \rightarrow A$  functional dependency holds ?

**Solution:**

- Find  $CG^+$
- IF  $A \in CG^+$  Then  
 $CG \rightarrow A$  holds  
ELSE  
 $CG \not\rightarrow A$  (means functional dependency do not hold)

**Exercise:**

- Is  $AC \rightarrow H$  holds ?
- Is  $AG \rightarrow I$  holds ?
- Is  $AI \rightarrow G$  holds ?

# Exercises

- Consider a Schema  $R = (A, B, C, D, E)$
- FDs  $F = \{A \rightarrow B, B \rightarrow C, C \rightarrow A, AB \rightarrow E, BD \rightarrow A\}$
- IS  $AC \rightarrow B$  holds ?                   **YES**
- IS  $C \rightarrow E$  holds ?                   **YES**
- IS  $AE \rightarrow D$  holds ?                   **NO**

### 3. Example for Computing F+

For each  $\gamma \subseteq R$ , we find the closure  $\gamma^+$ , and for each  $S \subseteq \gamma^+$ , we output a functional dependency  $\gamma \rightarrow S$ .

$$R = (A, B, C)$$

$$F = \{ A \rightarrow B, B \rightarrow C \}$$

All the subsets of R are -A,B,C,AB,AC,BC,ABC

Take one subset from above set say -A and find  $A^+$

$$A^+ = \{A, B, C\}$$

*Subsets of  $A^+$  are - A,B,C,AB,AC,BC,ABC*

Therefore following functional dependencies hold

$$A \rightarrow B, A \rightarrow C, A \rightarrow AB, A \rightarrow AC, A \rightarrow BC, A \rightarrow ABC. \quad \text{----(1)}$$

Similarly , take next subset say B and Find  $B^+$

$$B^+ = \{B, C\}$$

*Subsets of  $B^+$  are - B,C,BC*

Therefore following functional dependencies hold

$$B \rightarrow C, B \rightarrow BC \quad \text{----(2)}$$

Take subset C , Find  $C^+$

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

Therefore  $C \rightarrow C$  (similarly  $A \rightarrow A$  and  $B \rightarrow B$  holds ) holds ----(3)

# ....Example for Computing $F^+$

**Take AB**

$$AB^+ = \{A, B, C\}$$

Subsets of  $AB^+$  are - A, B, C, AB, AC, BC, ABC

Therefore following functional dependencies hold

$$AB \rightarrow B, AB \rightarrow C, AB \rightarrow AB, AB \rightarrow AC, AB \rightarrow BC, AB \rightarrow ABC \text{ ----(4)}$$

**Take BC**

$$BC^+ = \{B, C\}$$

Subsets of  $BC^+$  are - B, C, BC

Therefore following functional dependencies hold

$$BC \rightarrow B, BC \rightarrow C, BC \rightarrow BC \text{ ---- (5)}$$

**Take AC**

$$AC^+ = \{A, B, C\}$$

Subsets of  $AC^+$  are - A, B, C, AB, AC, BC, ABC

Therefore following functional dependencies hold

$$AC \rightarrow B, AC \rightarrow C, AC \rightarrow AB, AC \rightarrow AC, AC \rightarrow BC, AC \rightarrow ABC \text{ ----(6)}$$

# ....Example for Computing $F^+$

Take ABC

$ABC^+ = \{A, B, C\}$

Subsets of  $ABC^+$  are - A, B, C, AB, AC, BC, ABC

Therefore following functional dependencies hold

$ABC \rightarrow B$ ,  $ABC \rightarrow C$ ,  $ABC \rightarrow AB$ ,  $ABC \rightarrow AC$ ,  $ABC \rightarrow BC$ ,  $ABC \rightarrow ABC$ . ----(7)

From (1) (2) (3) (4) (5) (6) (7)

by eliminating duplicates we can get all functional dependencies set

$$F^+ = \{ A \rightarrow A, B \rightarrow B, C \rightarrow C, A \rightarrow B, A \rightarrow C, A \rightarrow AB, A \rightarrow AC, A \rightarrow BC, A \rightarrow ABC, B \rightarrow C, B \rightarrow BC, AB \rightarrow B, AB \rightarrow C, AB \rightarrow AB, AB \rightarrow AC, AB \rightarrow BC, AB \rightarrow ABC, AC \rightarrow B, AC \rightarrow C, AC \rightarrow AB, AC \rightarrow AC, AC \rightarrow BC, AC \rightarrow ABC, ABC \rightarrow B, ABC \rightarrow C, ABC \rightarrow AB, ABC \rightarrow AC, ABC \rightarrow BC, ABC \rightarrow ABC \}$$

# Exercises

- Consider a relation
  - gradeInfo (rollNo, studName, course, grade)
- Suppose the following FDs hold:
  - $\{rollNo, course\} \rightarrow \{grade\}$ ;
  - $\{studName, course\} \rightarrow \{grade\}$  ;
  - $\{rollNo\} \rightarrow \{studName\}$ ;
  - $\{studName\} \rightarrow \{rollNo\}$
- Is this a candidate Key (studName, course) ?