



SLIIT

Discover Your Future

INTRODUCTION TO DBMS & CONCEPTUAL DATABASE DESIGN



**COMPUTER
SYSTEMS
ENGINEERING**

Learning Outcomes (LO1)

- ▶ Understand what is a database management system
- ▶ Determine the importance of using a DBMS
- ▶ Understand the process of database designing
- ▶ Ability to design a conceptual database for a given scenario

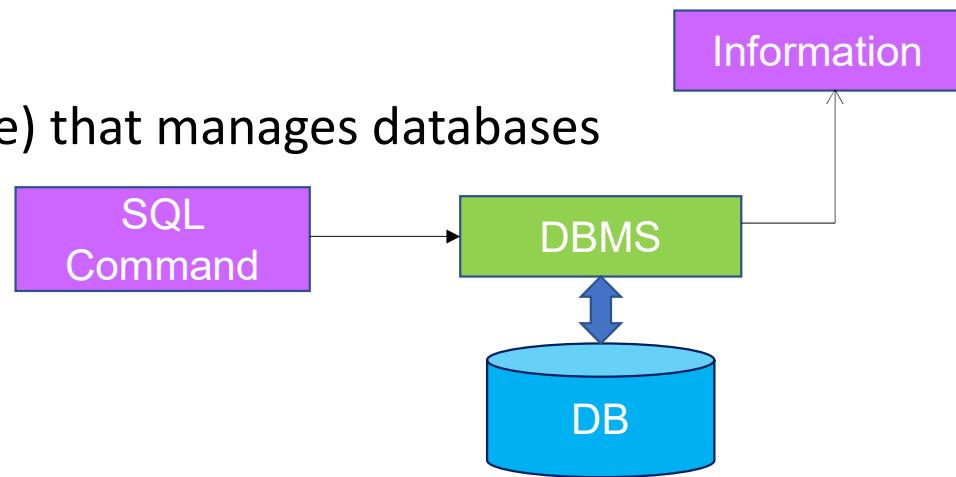


Topics

- Overview of DBMS
- Database Design process
- Revision basic ER modeling
- EER Concepts
 - Inheritance – IS-A Relationship
 - Aggregation
- How to develop an EER
- Exercises

What is a DBMS?

- A database is a collection of related data
- DBMS is a general purpose software system that facilitates the processes of defining, constructing, manipulating, and sharing databases among various users and applications
- Simply, DBMS is a piece of software (package) that manages databases
 - Store data
 - Retrieve data

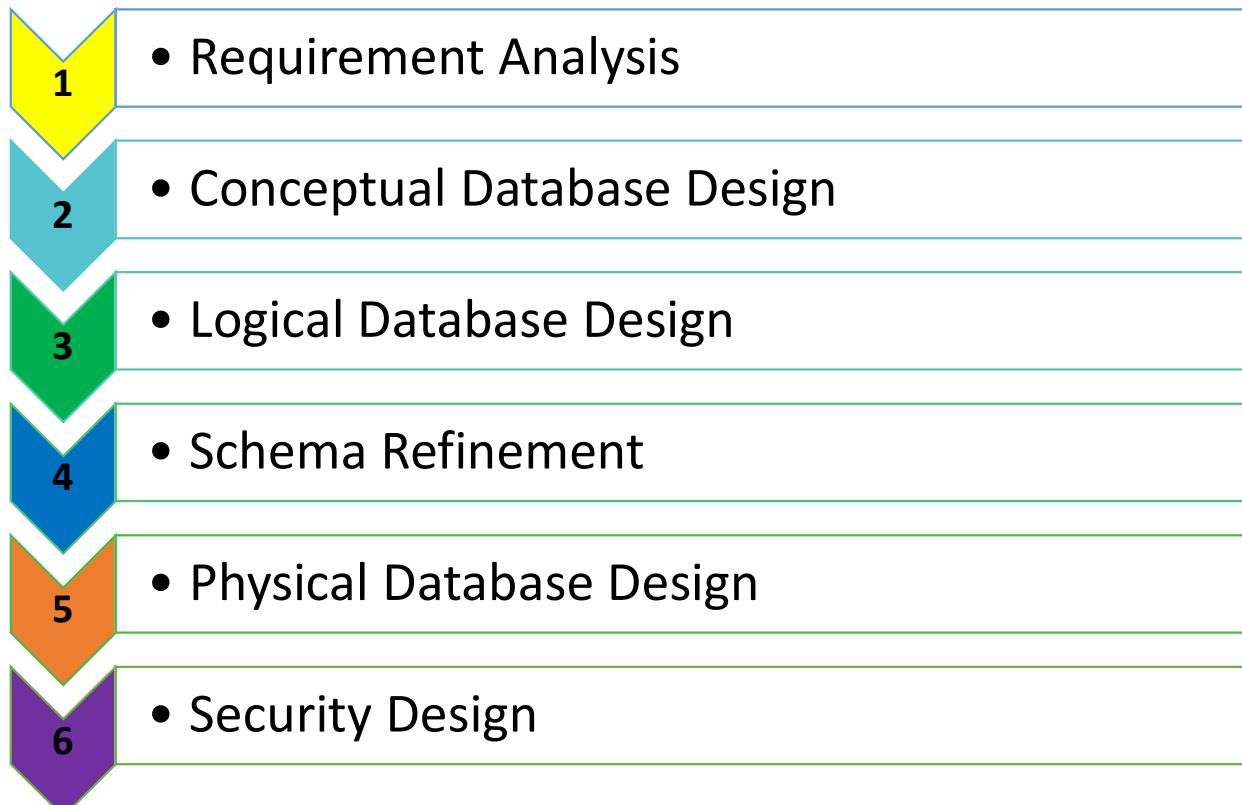


Why a DBMS?

- Large storage
- Data independence
- Efficient data access
- Data integrity
- Security
- Easy data administration
- Concurrent transaction processing

W'hy
Choose

Database Design Process



1

• Requirement Analysis

- Purpose -> Gather and analyze the data requirements
- How -> Interviews, Review documents/existing systems
- Questions -> What to store?
 - Relationships among objects?
 - Types of queries?
 - Types of users?
 - Performance/ security/ administrative constraints?
- Output -> Concisely written set of user requirements

• Conceptual Database Design

- High level conceptual model is designed encapsulating the user requirements
- Entity Relationship (ER) Model is the widely used conceptual model
- ER Model is enhanced with object oriented concepts (Inheritance and Aggregation) and known as the Enhanced Entity Relationship (EER) Model



• Relational Model Terminology

- A *relation* is a table with columns and rows.
 - Only applies to logical structure of the database, not the physical structure.
- An *attribute* is a named column of a relation.
- The *degree* of a relation is the number of attributes in a relation.
- A *domain* is the set of allowable values for one or more attributes.



• Relational Model Terminology

- A *tuple* is a row of a relation.
- The *cardinality* of a relation is the number of tuples in a relation.
- A *relational database* is a collection of relations with distinct relation names.
 - The relations are usually normalized
 - Sometimes there are other kinds of objects in a relational database besides relations.

• Relational Model Terminology

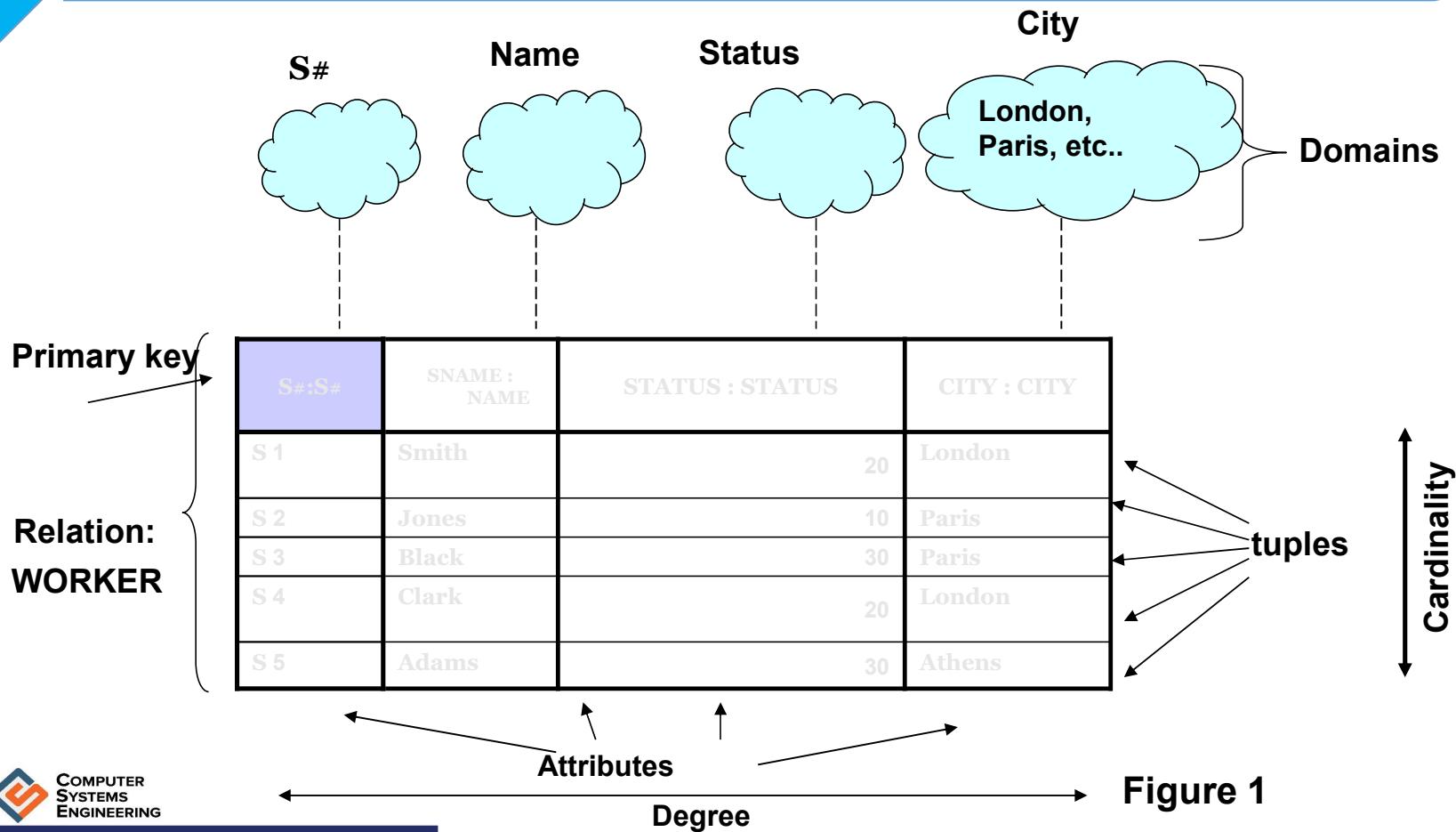
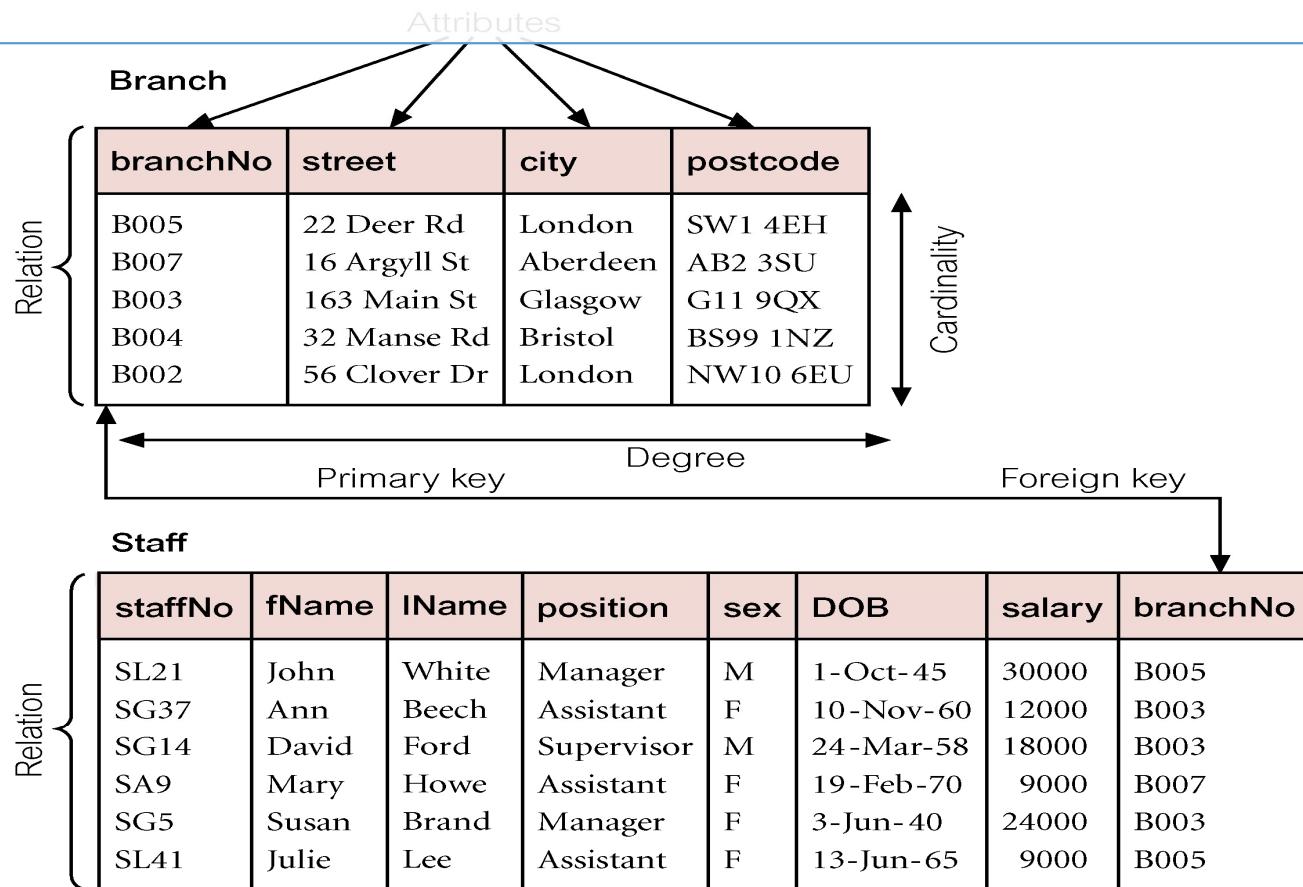


Figure 1

• Relational Model Terminology

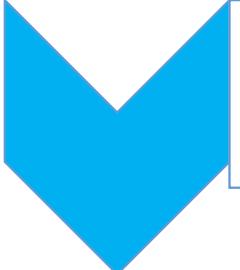


• Relational Model Terminology

Attribute	Domain Name	Meaning	Domain Definition
branchNo	BranchNumbers	The set of all possible branch numbers	character: size 4, range B001–B999
street	StreetNames	The set of all street names in Britain	character: size 25
city	CityNames	The set of all city names in Britain	character: size 15
postcode	Postcodes	The set of all postcodes in Britain	character: size 8
sex	Sex	The sex of a person	character: size 1, value M or F
DOB	DatesOfBirth	Possible values of staff birth dates	date, range from 1-Jan-20, format dd-mmm-yy
salary	Salaries	Possible values of staff salaries	monetary: 7 digits, range 6000.00–40000.00

Concepts of a ER Model

- Entity types
- Relationship types
- Attributes



Entity Types

- Entity type
Group of objects with the same properties, identified by the enterprise as having an independent existence.
E.g. The notebook computer entity type
- Entity occurrence or entity instance
A uniquely identifiable object of an entity type.
E.g. The notebook computer on which I am now typing

Physical existence

Staff	Part
Property	Supplier
Customer	Product

Conceptual existence

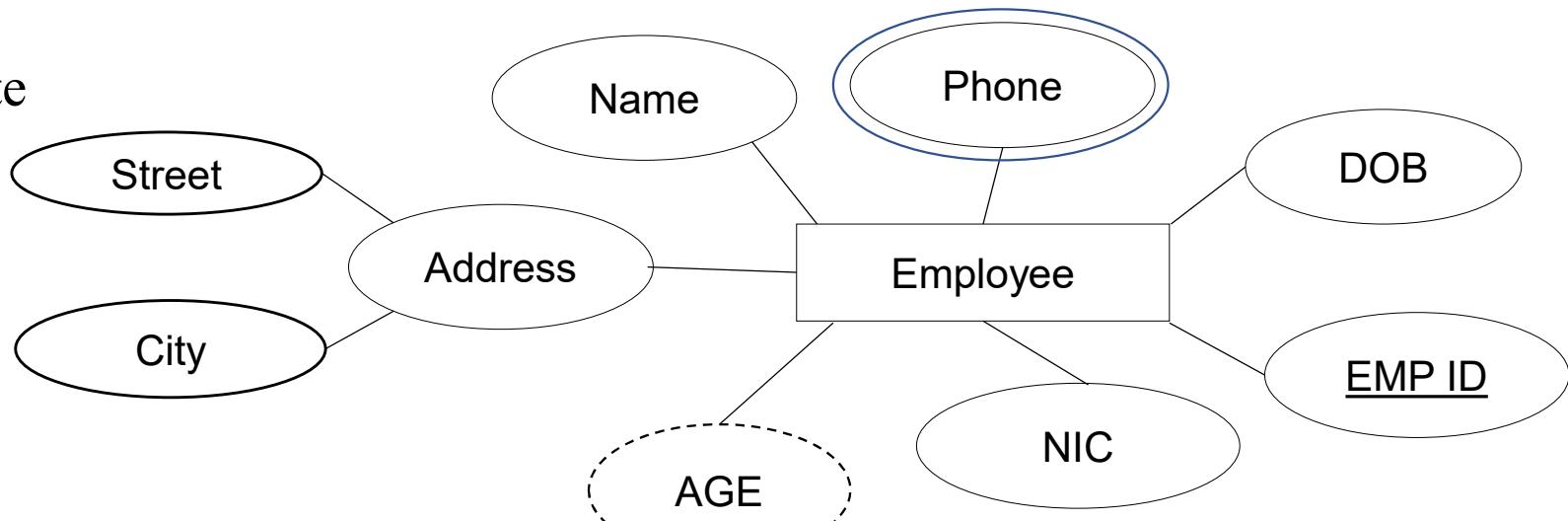
Viewing	Sale
Inspection	Work experience

Attributes and attribute domains

- **Attribute**
Property of an entity or a relationship type.
- **Attribute Domain**
Set of allowable values for one or more attributes.

• Entities, Attributes & Keys

- Simple Attribute
- Multivalued Attribute
- Composite Attribute
- Derived Attribute



Relationship Types

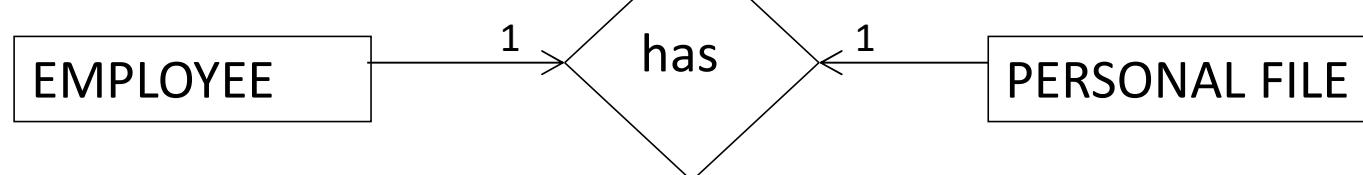
- **Relationship type**
Set of meaningful associations among entity types.
- **Relationship occurrence**
Uniquely identifiable association, which includes one occurrence from each participating entity type.

The Degree of a Relationship

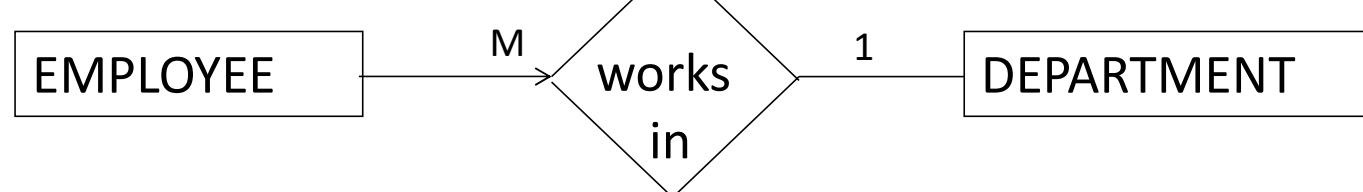
- Degree of a Relationship is the number of entities participating in the relationship.
 - A relationship between two entities is called a *binary* relationship
 - three is *ternary*
 - four is *quaternary*.

• Binary Relationships

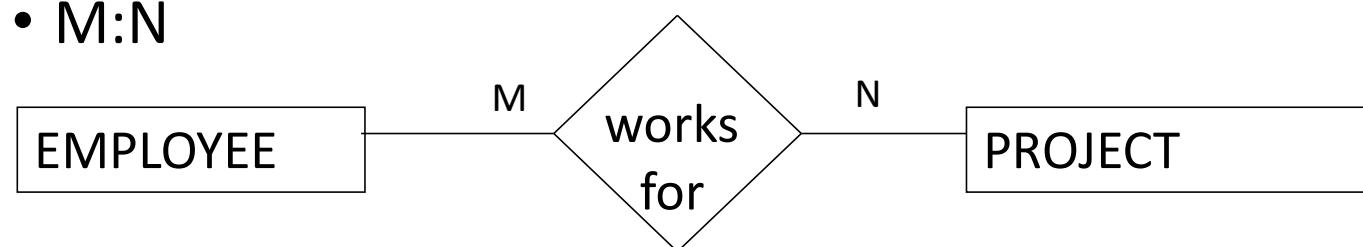
- 1:1



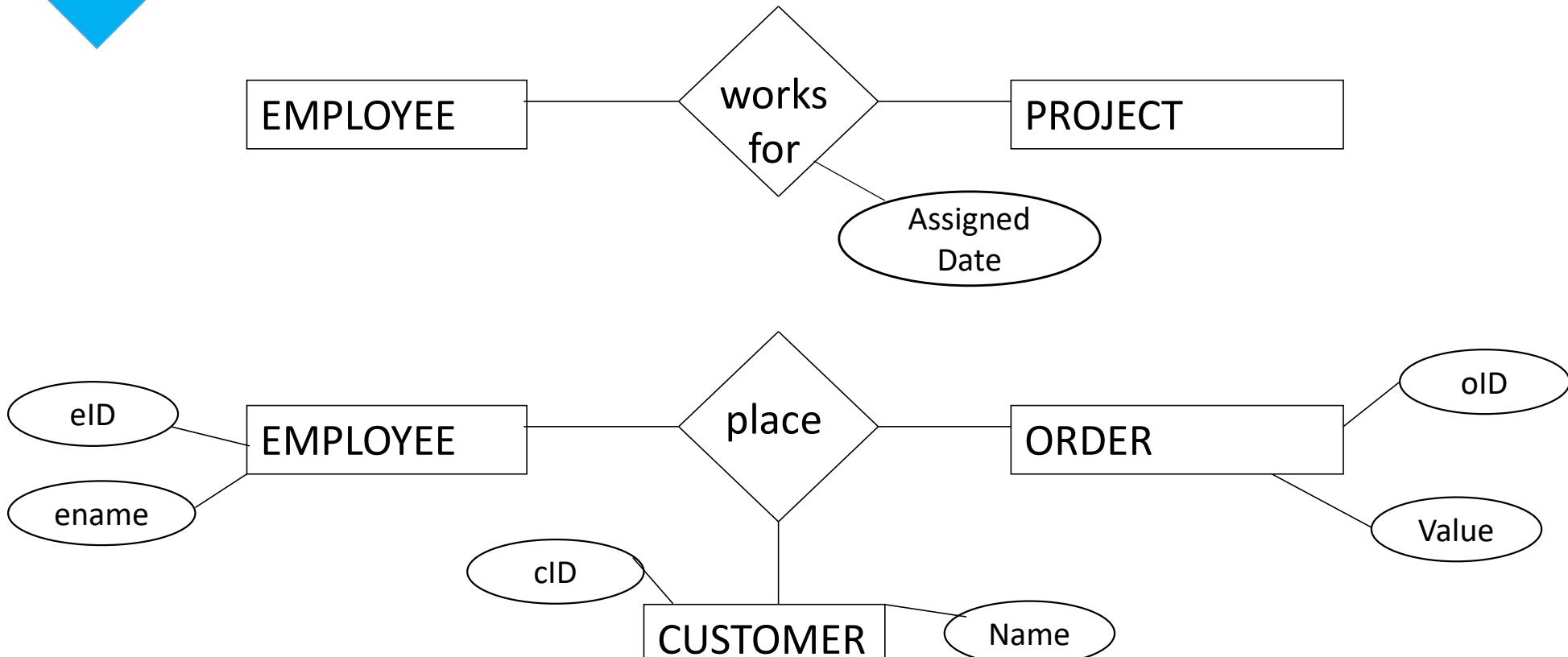
- M:1



- M:N



• Relationships

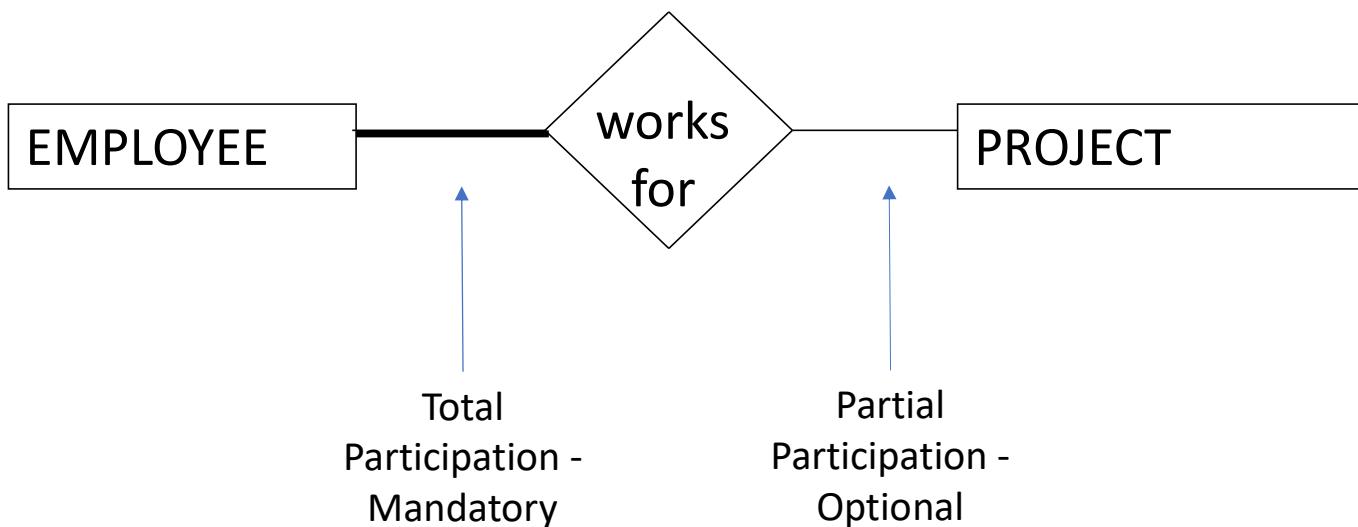




• Structural Constraints

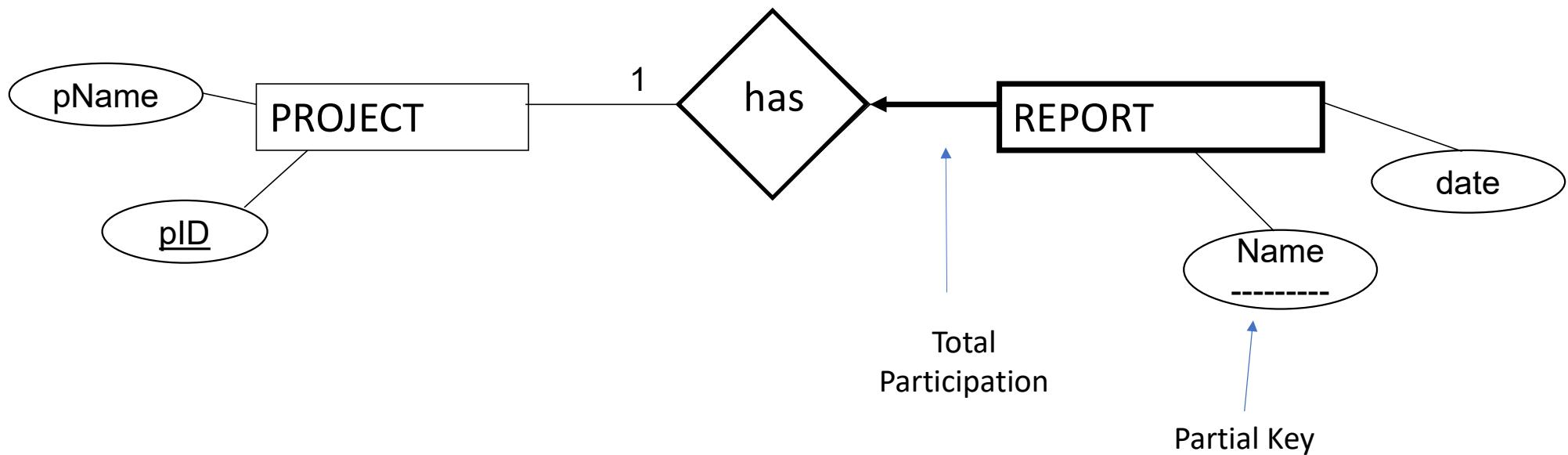
- Main type of constraint on relationships is called *multiplicity*.
- Multiplicity - number (or range) of possible occurrences of an entity type that may relate to a single occurrence of an associated entity type through a particular relationship.
- Represents policies (called *business rules*) established by user or company.
- Binary relationships are generally referred to as being:
 - one-to-one (1:1)
 - one-to-many (1:*)
 - many-to-many (*:*)

• Participation Constraints



An employee must work for a project

• Weak Entities



• ISA Relationship

- Specialization

- Process of defining sub classes from an entity type

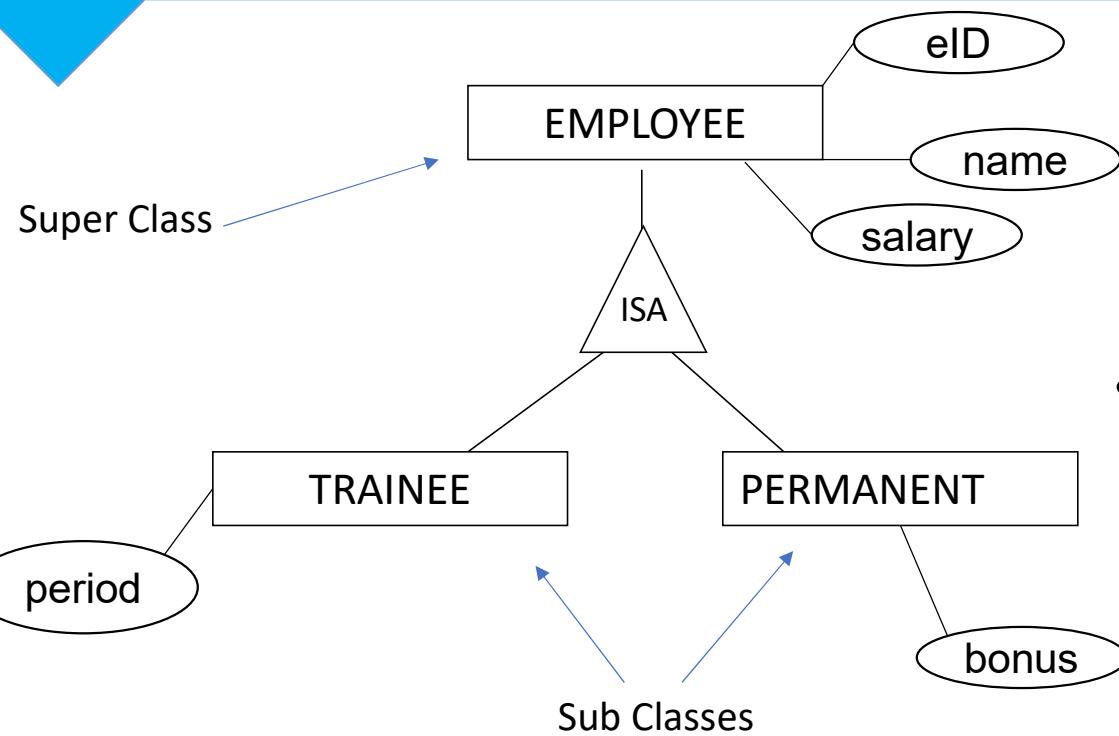
- Generalization

- Process of identifying commonalities between entity types and grouping them as super classes

Top Down Approach

Bottom Up Approach

• ISA Relationship



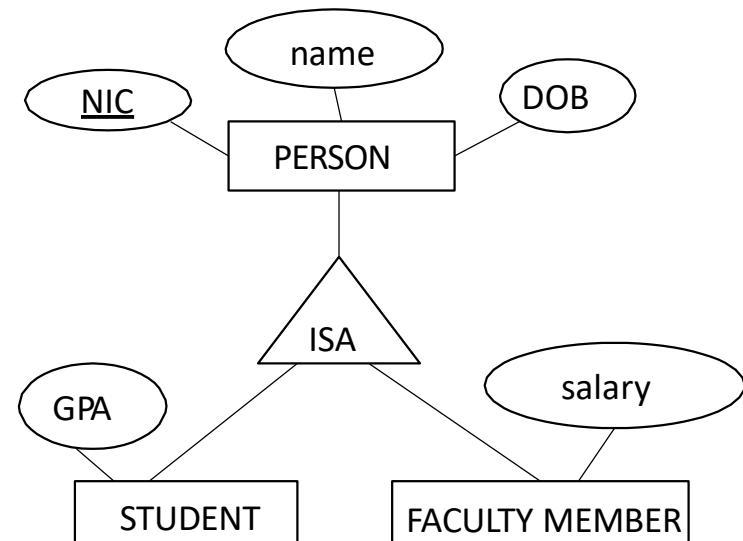
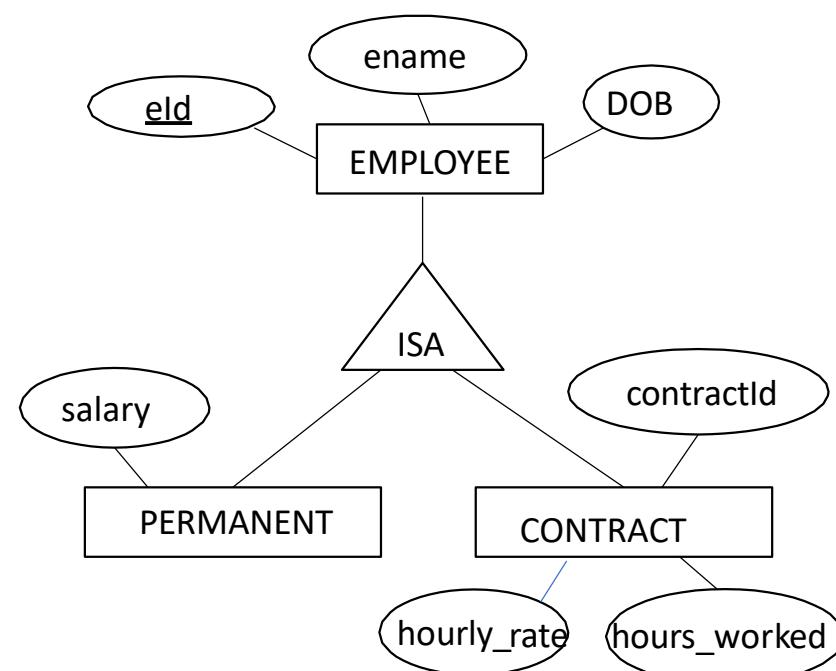
- Why create subclasses?
 - There may be attributes in the sub-class which are not in the superclass
 - There may be some relationship types that apply only to the sub-classes



• Disjointness Constraint

- By default, two subclasses are considered DISJOINT (An entity cannot belong to more than one subclass)
 - Can an employee be a permanent employee and a contract employee? NO! Therefore, the permanent employee subclass and the contract employee subclass are **disjoint**.
- **Overlap Constraints** determine whether two subclasses are allowed to contain the same entity
 - Can a person in a university be a student and a faculty member at the same time? YES! If it is so, we denote this by writing **student overlaps faculty member**. In absence of such a statement we assume that the sub classes are disjoint.

• Disjointness Constraint



Student overlaps Faculty Member

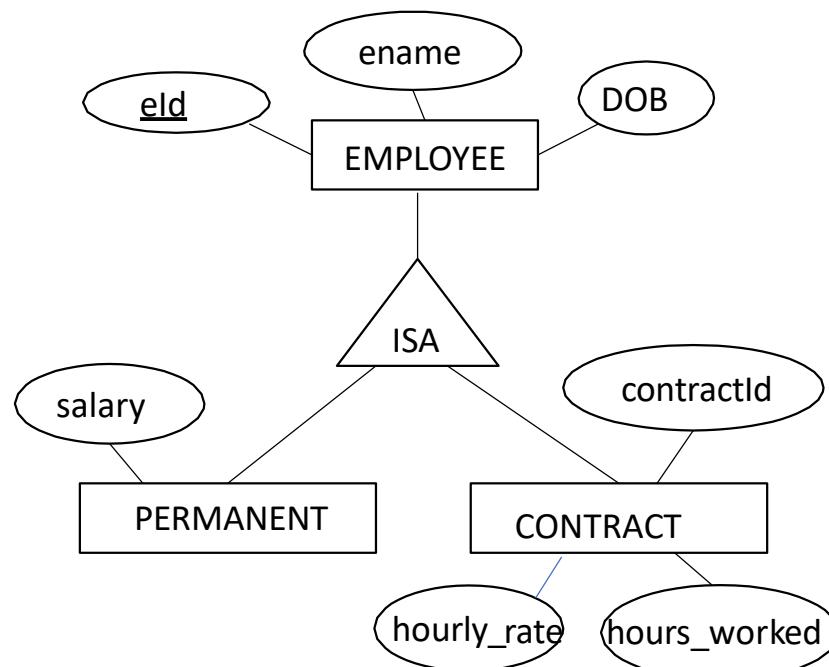
Permanent and Contract sub classes are disjoint



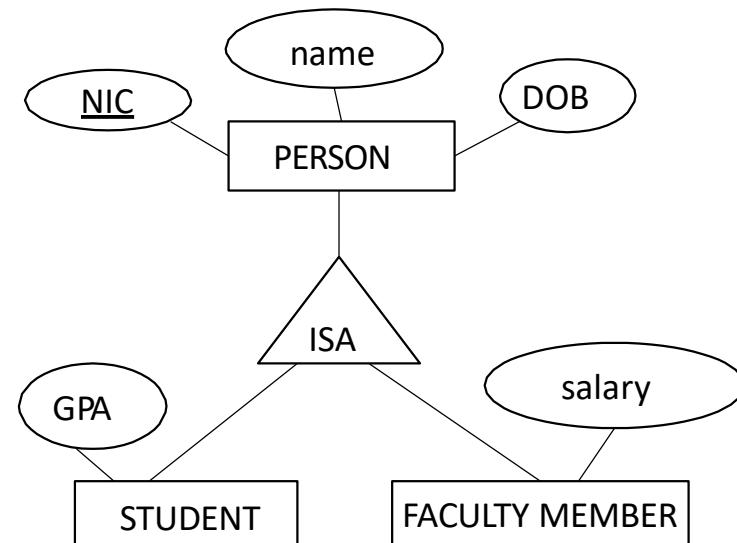
• Completeness Constraint

- **Covering Constraints** determine the entities in the subclass collectively include all the entities in the superclass
 - Does every employee belong to either permanent employee or a contract employee? YES! If it is so, we denote this by writing **permanent employee and contract employee covers employee**.
 - Does every person in a university belong to either student and a faculty member? NO! Therefore, there is no covering constraint.
- Existence of a covering constraint is also known as having a Total Specialization
- Absence of a covering constraint is known as having a Partial Specialization

• Completeness Constraint



Permanent Employee and Contract Employee Covers Employee



Student and Faculty Member does not cover Person

Options

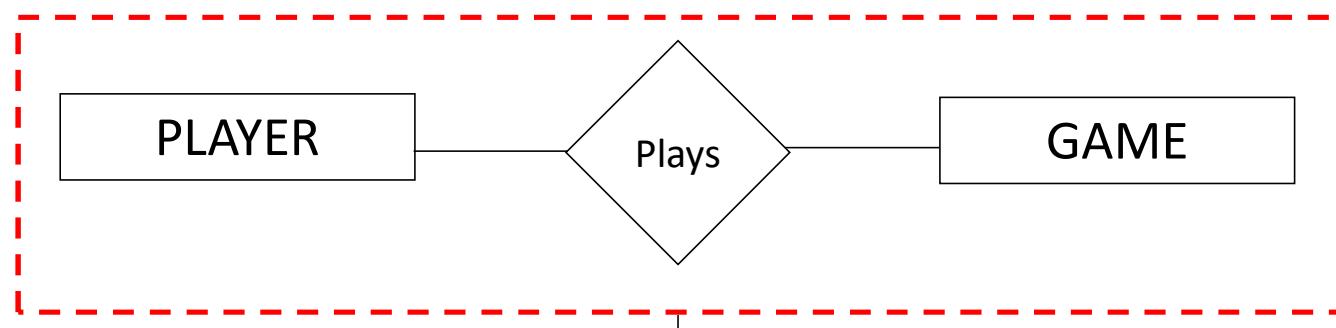
	Method	Overlapping	Disjoint	Total	Partial
Option 1 (Multi Relation)	Create additional separate tables for subclasses with same primary key	YES	YES	YES	YES
Option 2 (Multi Relation)	Create tables for just the subclasses	YES	YES	YES	NO
Option 3 (Single Relation)	Have table for superclass only with single type attributes	MAYBE	YES	YES	MAYBE
Option 4 (Single Relation)	Have table for superclass only with separate (Boolean) type attribute for each subclass	YES	YES	YES	YES



• Aggregation

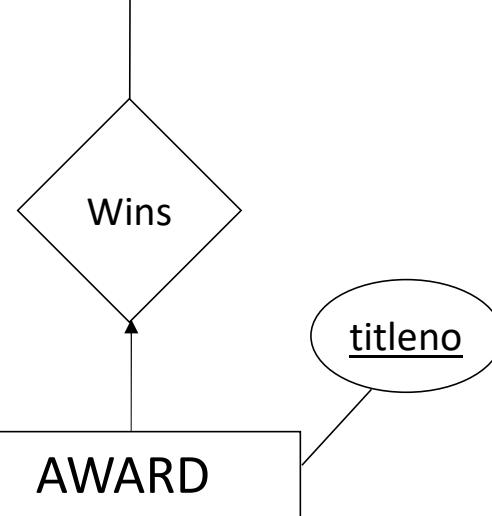
- Sometimes it is possible for a relationship to participate in another
- Difference between ternary and aggregation is that **aggregation** contains **two independent relationships**, whereas in ternary there is only one

• Aggregation



Imaginary Entity

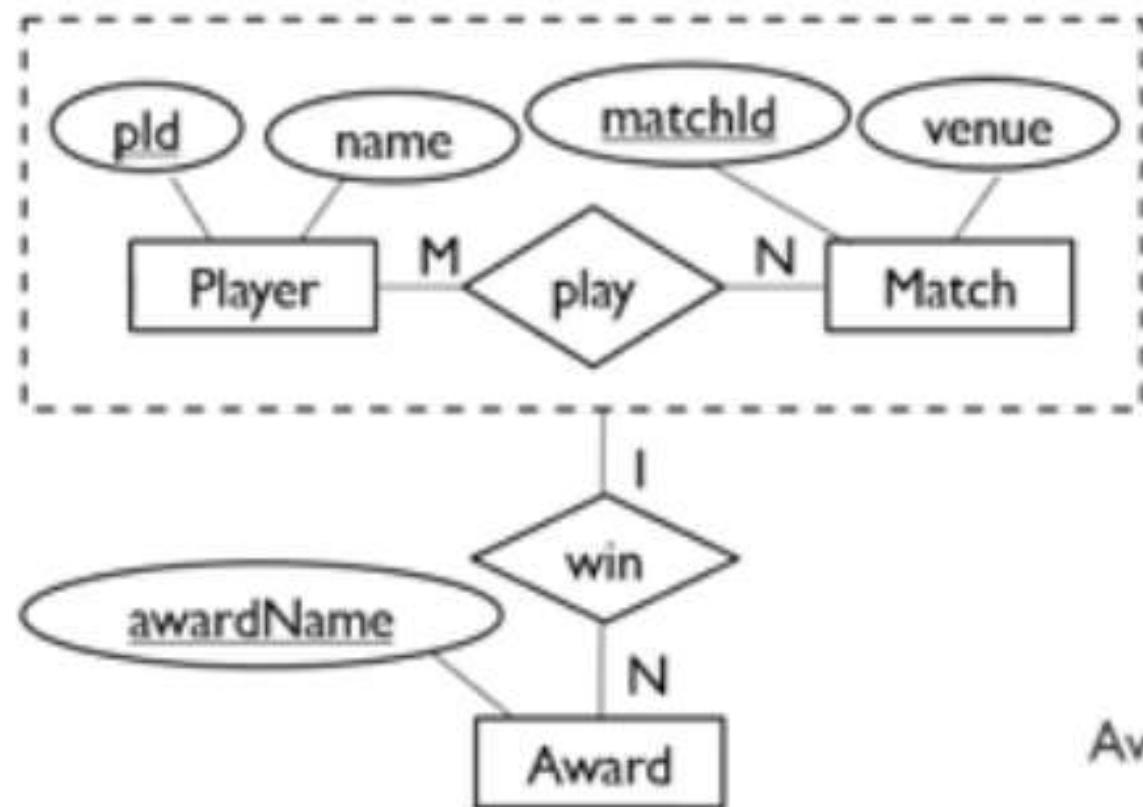
Player plays games. Based on the performance of the match, player wins an award.



Process

- Step 1 - First map relationship R as a new entity
- Step 2 – To map the aggregate relationship treat the R entity as a normal entity and use the primary key of R in the mapping
- **Important** – the cardinality of each relationship can be any of the possible options

Example



Player (pld, name)

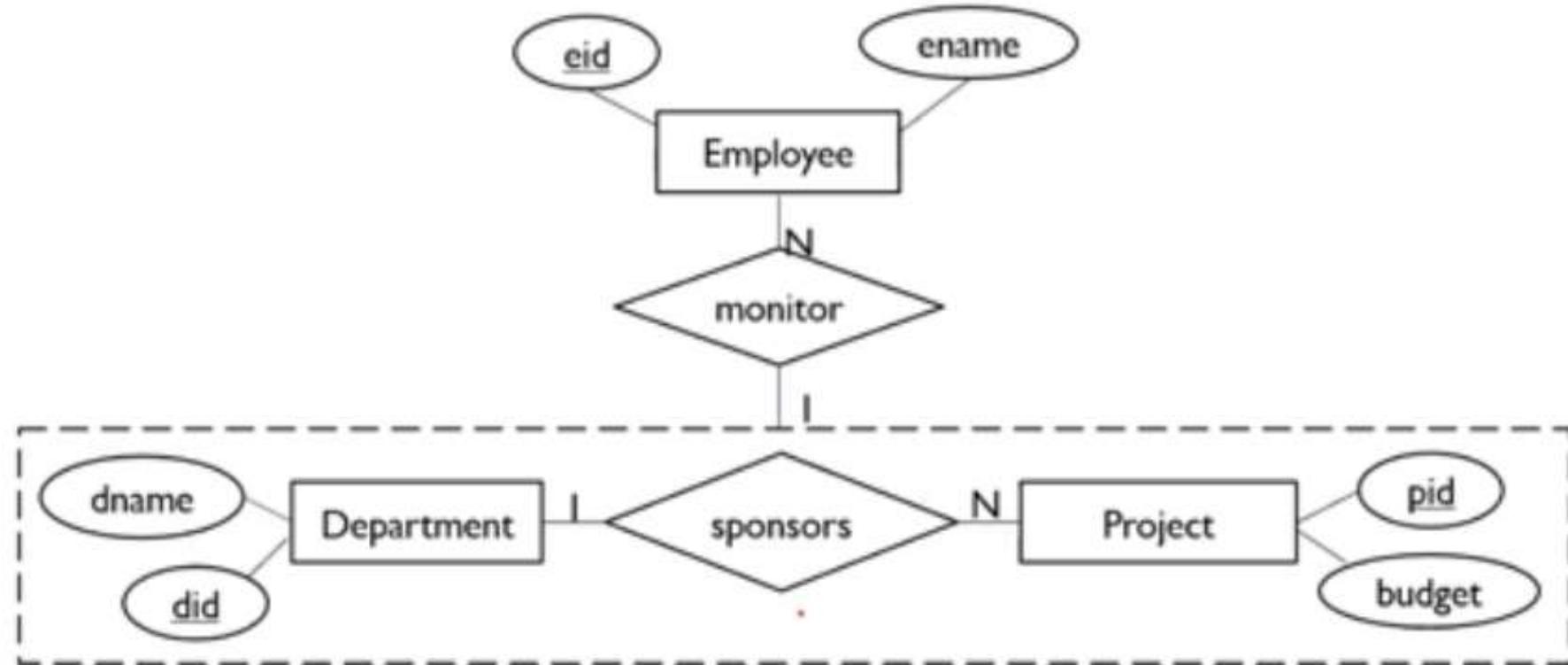
Match (matchId, venue)

Play (pld, matchId)

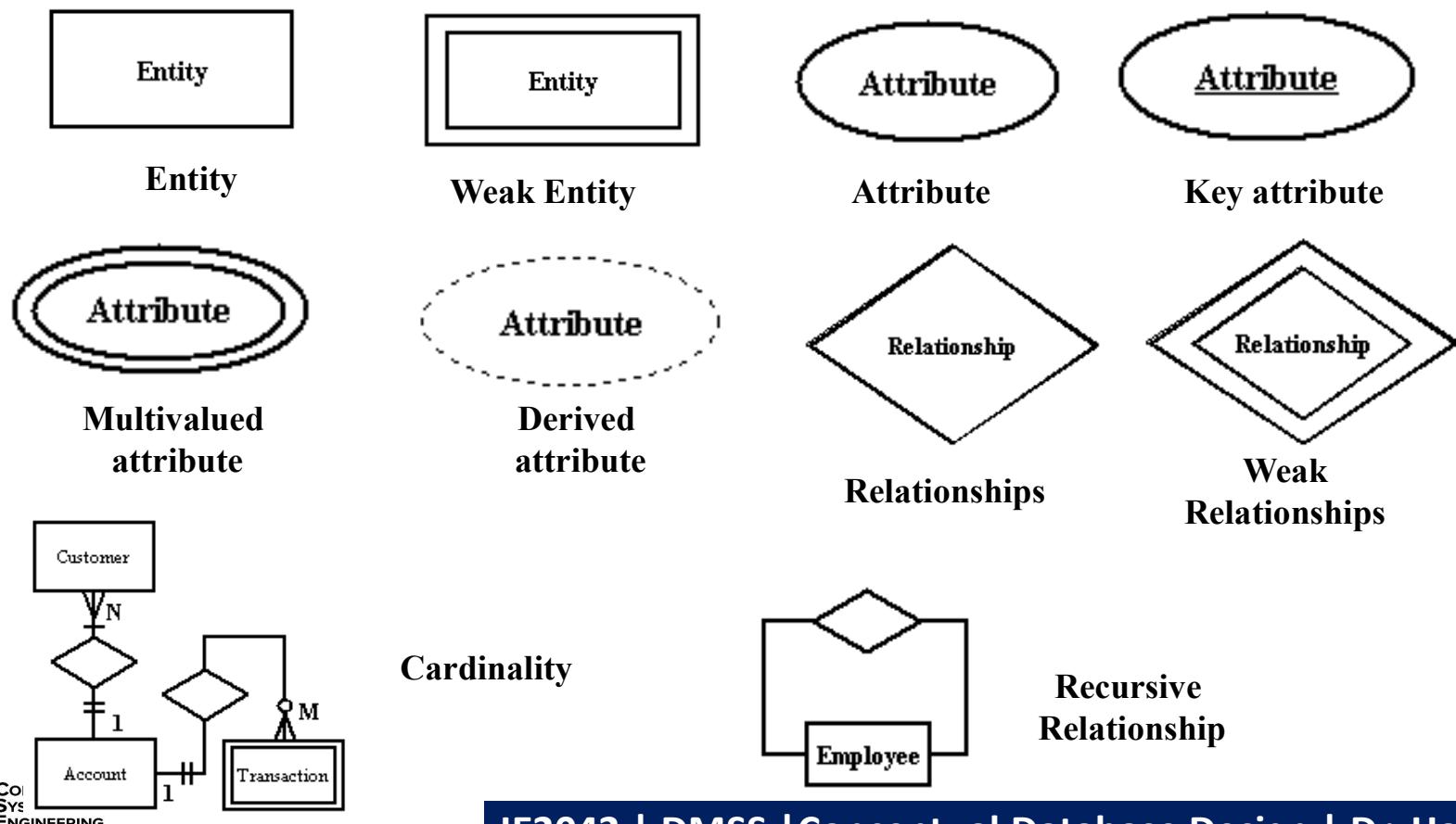
Award (awardName, pld, matchId)

Exercise

- Map the following EER diagram to relational model.



• Chen Notation





• How to build an ER Model

- **Identify data objects** (entities) about which the system needs to store data.
- For each entity, **list its attributes** and **check the attributes for completeness**.
- Using the practical guidelines listed below, **make any necessary improvements**.
 - **Data items** should be **put into logical groups** – groups that go together.
 - For each data group, or **entity**, there should be a **key that uniquely identifies individual members** of that entity



• How to build an ER Model

- There should be **no redundant data** in the model – data is deemed redundant in the following situations:
 - The data is never used by the system
 - The same data items (customername, customeraddress) are stored in more than one place in the system.
 - Data in one place can be derived from data held in another place in the system.



• How to build an ER Model

- Investigate and **record relationships**.
 - link entities so that all significant real-life relationships are captured.
- **Check the entity descriptions against the data dictionary descriptions**
 - Make sure that each process on the data flow diagram has available to it all the data it needs;
 -  Britton, C., Doake, J., *Chapter 5: Data Modelling*, Software System Development – A gentle introduction, McGraw-Hill

Activity



- Draw an EER diagram for the following requirements.
 - Students contain an id (unique), name and an address.
 - There are academic semesters containing a *semester id* (unique), *semester* and *year*.
 - There are courses offered during academic semesters. A course has a *number* (unique), *name* and *credits*.
 - Students make payments. A payment has *receipt number* (unique), *amount* and *date*.
 - Payments can be classified into Tuition (semester payment), Examination and Other (Library fine, Printouts).
 - A Tuition payment is made for an academic semester.
 - For other payments description should be stored.
 - Students register for courses offered during a particular semester. The registered date must be stored in the database.



SLIIT

Discover Your Future

LOGICAL DATABASE DESIGN



**COMPUTER
SYSTEMS
ENGINEERING**

IE2042 | DMSS | Logical Design | Dr. Harinda Fernando

Learning Outcomes (L01)

- ▶ Understand the process of logical database design
- ▶ Understand the relational model and its components
- ▶ Explain the integrity constraints
- ▶ Ability to convert an EER diagram to a relational model



The Relational Model

- A **relational database** is a collection of relations with distinct relation names
- The main construct of the relation model is the **relation**
- A relation consist of;
 - **Relational Schema**
 - **Relational Instance**

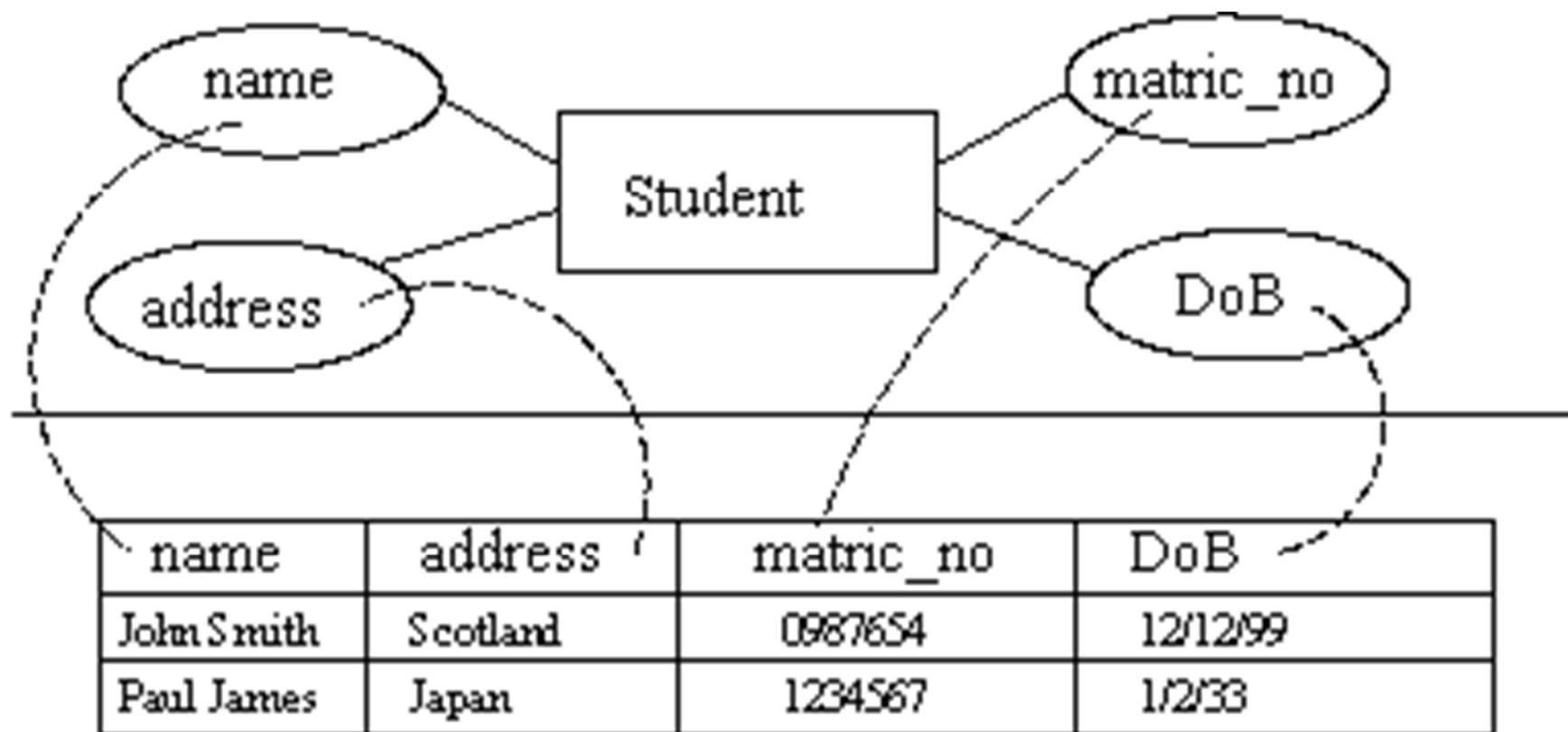
Relation Mapping

Mapping ER models to relations

What is a relation?

- A relation is a table that holds the data we are interested in. It is two-dimensional and has rows and columns.
- Each entity type in the ER model is mapped into a relation.
 - The attributes become the columns.
 - The individual entities become the rows.

Example



The Relational Model

- Relation Schema = **Relation Name + Each Field Name + Each Field's Domain**
Student (sID:integer , sName:varchar(20) , Age:integer)
- Relation Instance = Set of tuples/ Records in the database

sID	sName	Age
1356	Kasun	22
6832	Rayan	20

Example

- Relations can be represented textually as:
tablename(primary key, attribute 1, attribute 2, ... , *foreign key*)
- If roll_num was the primary key, and there were no foreign keys, then the table above could be represented as:
student(roll_num, name, address, date_of_birth)

Integrity Constraints

- Integrity Constraint is a rule/ condition enforced by the DBMS to ensure only correct data is stored in the database
- If an instance satisfies all the integrity constraints specified on the database schema; then the instance is **Legal**
- Types of integrity constraints are;
 - **Domain Constraints**
 - **Key Constraints**
 - **Referential Constraints**
 - **Other Constraints**

Domain Integrity Constraints

- Specifies that the values in a column must be from the domain specified
- Data types are used to validate the domain
 - GPA should be a float

Key Integrity Constraints

- Unique and Primary key
- The minimum set of attributes that uniquely identifies a tuple = **KEY**
 - Cannot insert many records to the Student table from one sID

Referential Integrity Constraints

- Enforced by Foreign Keys
- Specified between two relations to maintain consistency among tuples in the two relations
- A tuple in one relation that refers to another relation must refer to an existing tuple in that relation
 - Both domains should be equal
 - Should refer existing values

Other Integrity Constraints (Table & Assertions)

- **Table Constraints**

- Constraints within a single table
- Use CHECK constraints

The account balance of the account table should be always greater than zero

(`AccTable-Balance>0`)

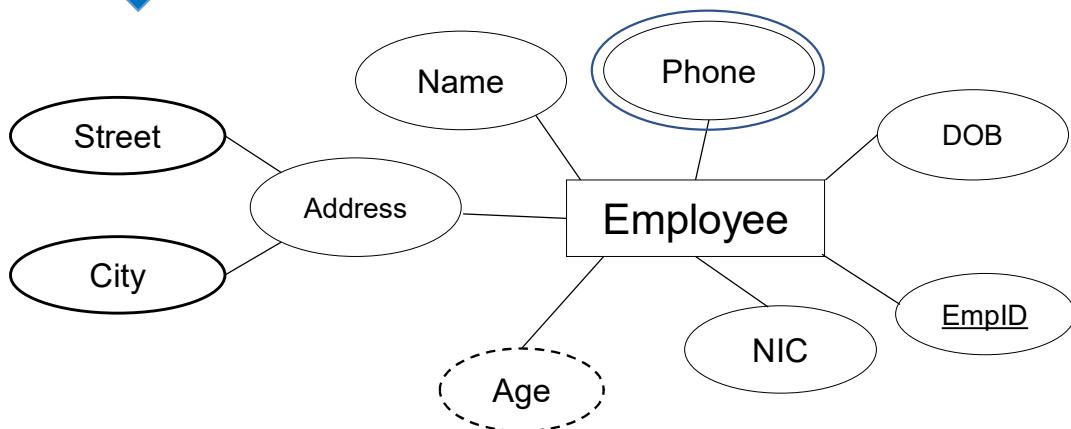
- **Assertions**

- Constraints between multiple tables
- Use Triggers

The employee salary should not exceed his managers salary

(`EmpTable`- see salary details & `DeptTable`- find his manager)

• Entities, Attributes & Keys



Employee ()

Employee (NIC, Name, DOB)

Employee (EmplID, NIC, Name, DOB)

Employee (EmplID, NIC, Name, DOB, Street, City)

Employee_Phone (EmplID, Phone)

Logical Mapping For Simple Relationships

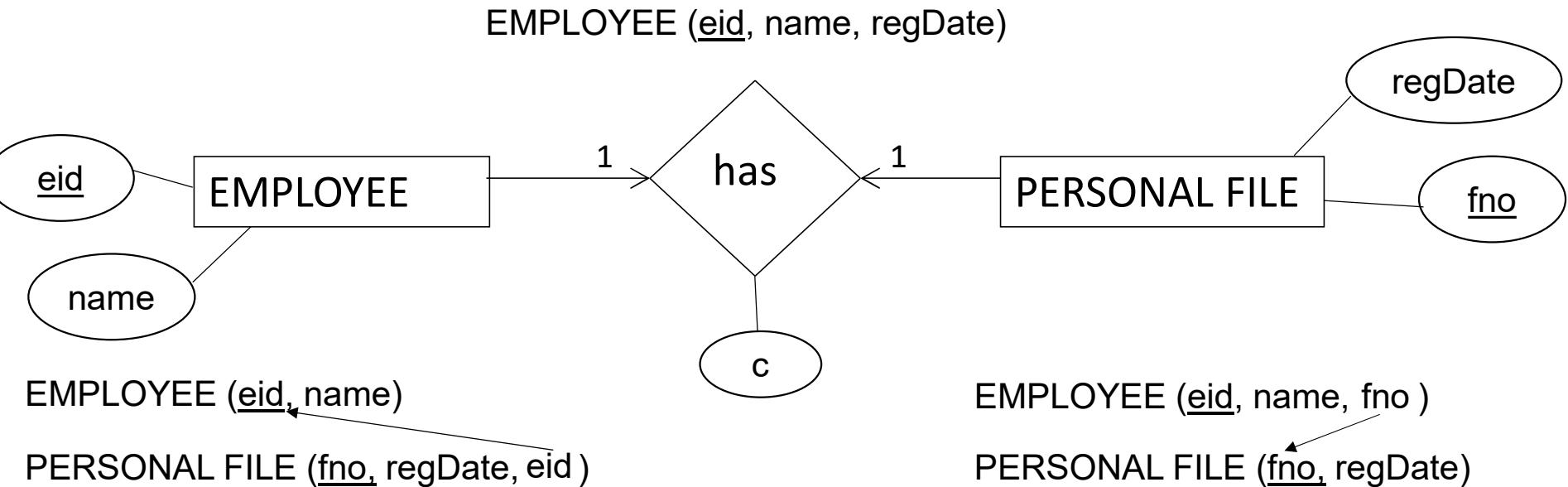
Mapping 1:1 relationships

- Before tackling a 1:1 relationship, we need to know its optionality.
- There are three possibilities the relationship can be:
 - mandatory at both ends
 - mandatory at one end and optional at the other
 - optional at both ends

Mandatory at both ends

- If the relationship is mandatory at both ends it is often possible to subsume one entity type into the other.
 - One of the keys are used as primary key
 - Combined table contains all attributes from both tables with duplicates removed
- Else the tables can be kept separate and connected using a foreign key

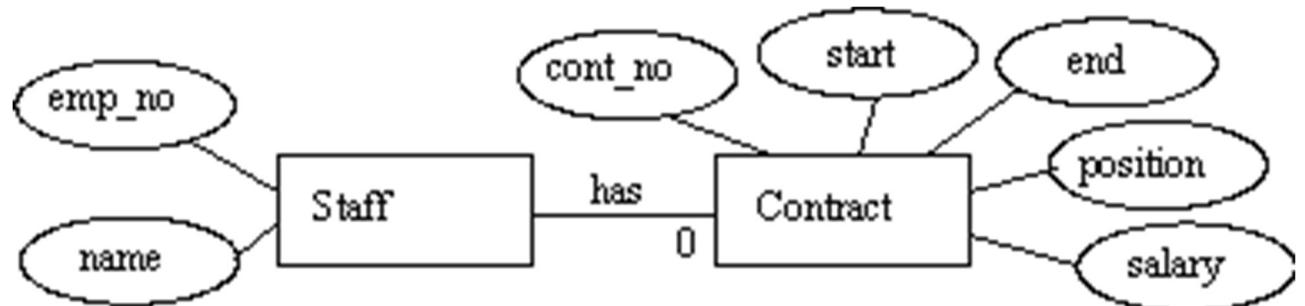
Mandatory at both ends



If there is a descriptive attribute -> It is included in the foreign key side

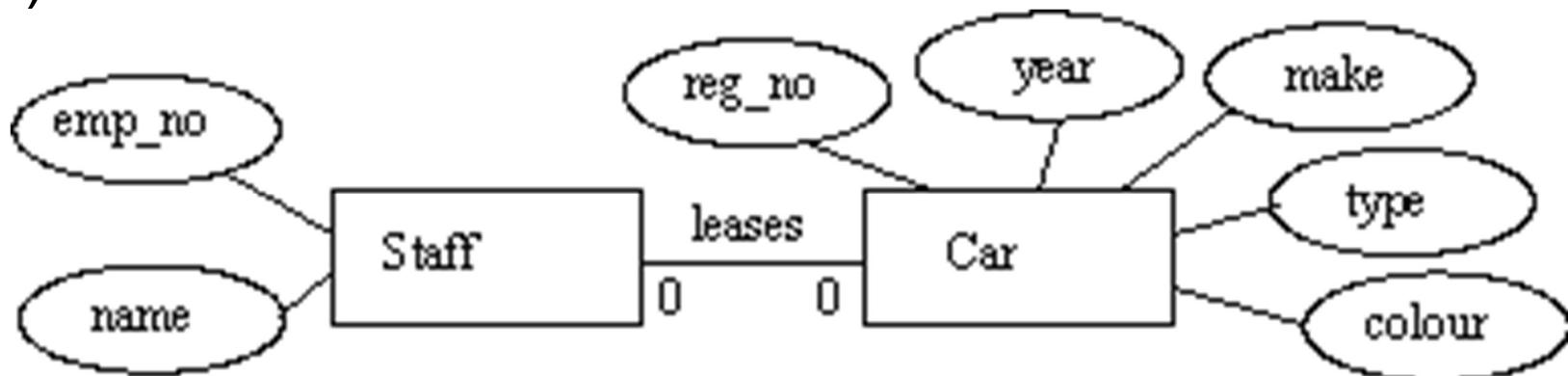
Mandatory <-> Optional

- The entity type of the optional end can be subsumed into the mandatory end as in the previous example.
- But should not be integrated as this will create null entries.
- It is better to include the foreign key in the optional side so as to avoid null entries.



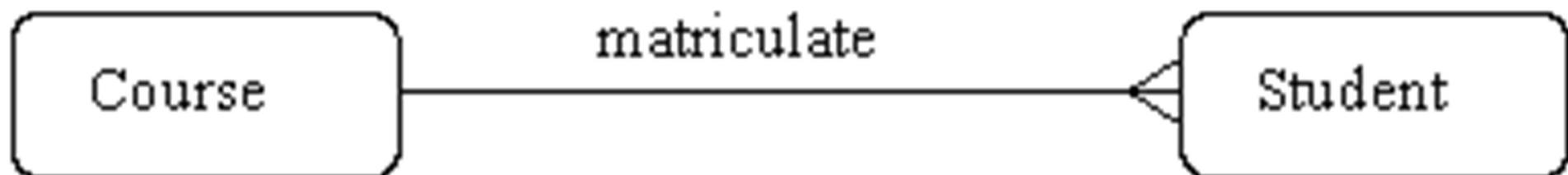
Optional at both ends...

- Such examples **cannot be amalgamated** as you could not select a primary key.
- Instead, one foreign key is used as before (which table is your choice).

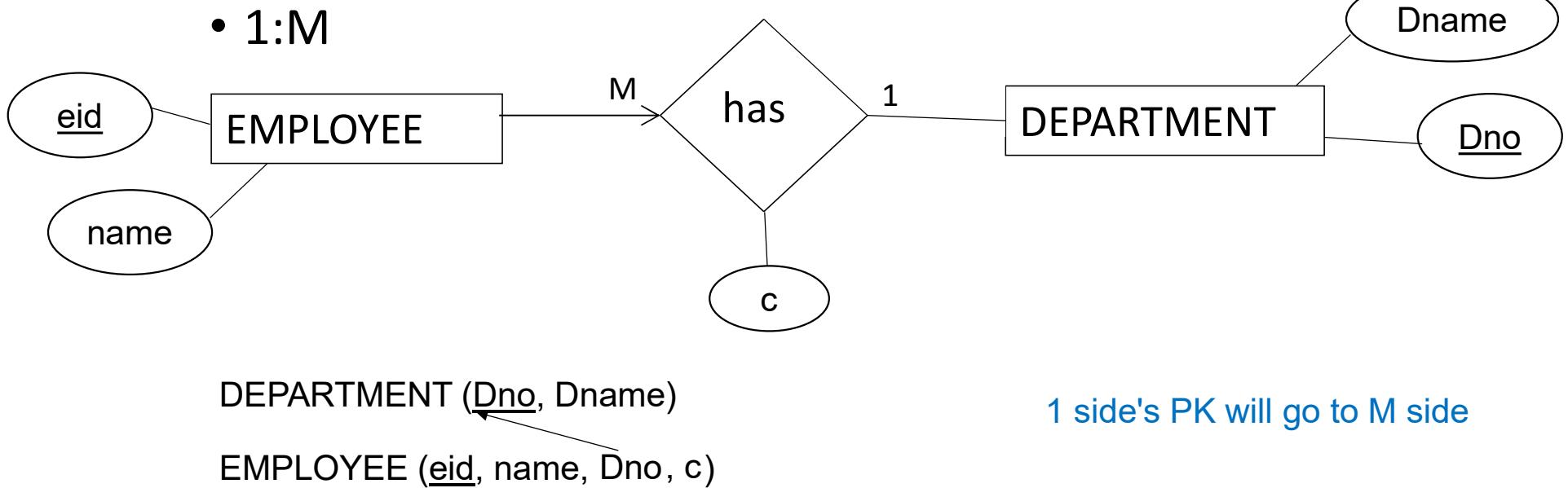


Mapping 1:m relationships

- Cannot be amalgamated
- To map 1:m relationships, the primary key on the ‘one side’ of the relationship is added to the ‘many side’ as a foreign key.
- For example, the 1:m relationship ‘course-student’:



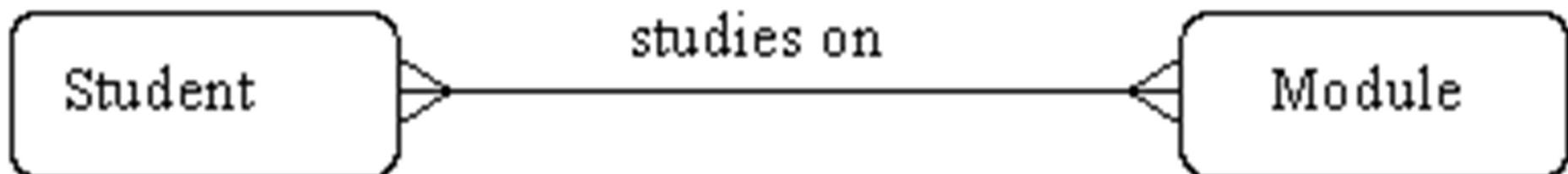
Mapping 1:m relationships



Even if total participation is available, no priorities will come in to play.
Schema will not change

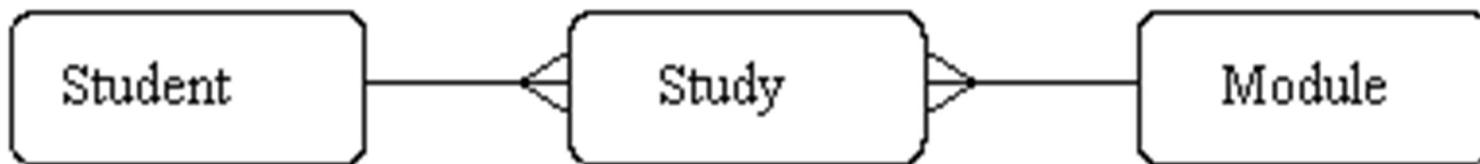
Mapping n:m relationships

- If you have some m:n relationships in your ER model then these are mapped in the following manner.
 - A new relation is produced which contains the primary keys from both sides of the relationship
 - These primary keys form a composite primary key.



Mapping n:m relationships

- This is equivalent to:

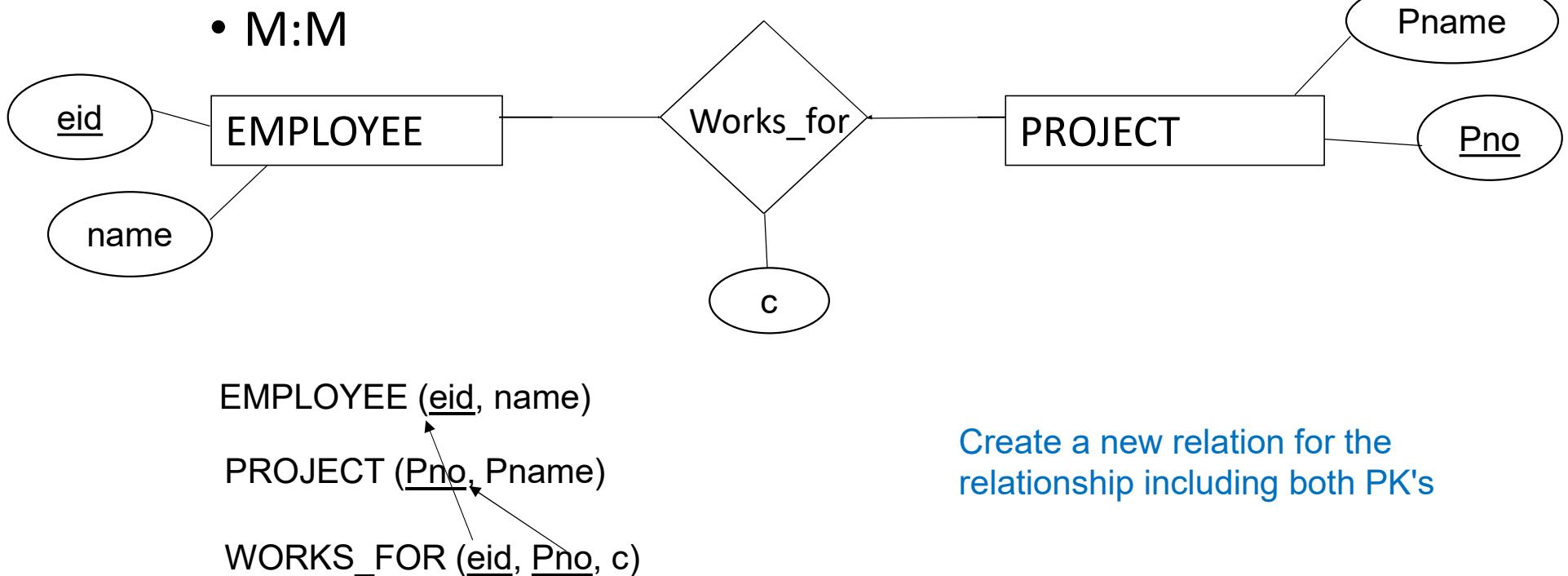


Student(matric_no,st_name,dob)

Module(module_no,m_name,level,credits)

Study()

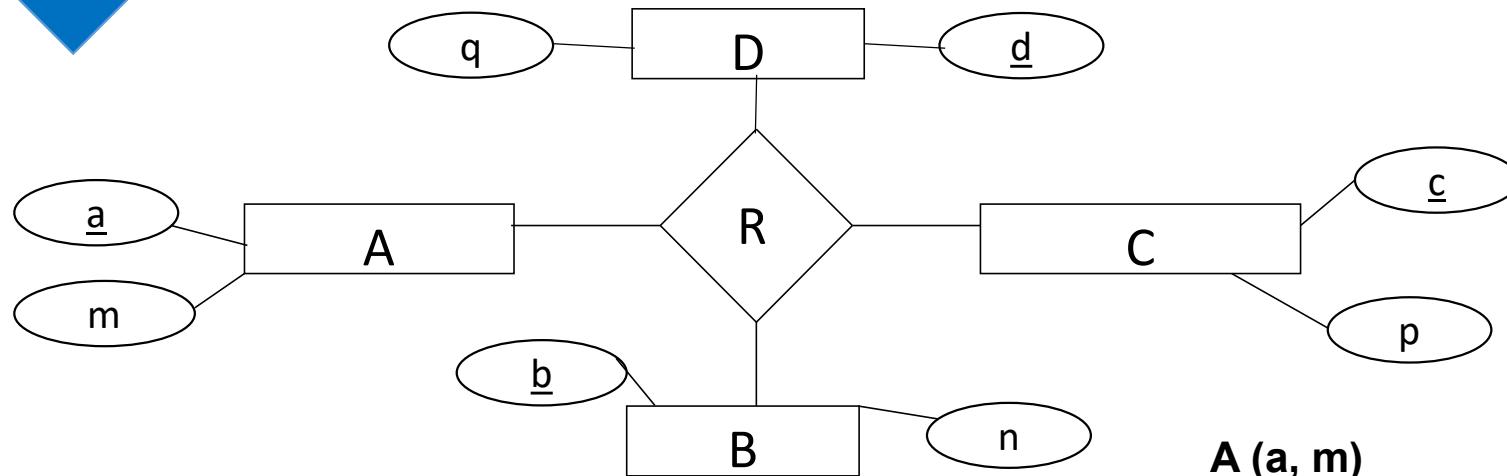
Mapping n:m relationships



Summary

- **1-1 relationships**
 - Depending on the optionality of the relationship, the entities are either combined or the primary key of one entity type is placed as a foreign key in the other relation.
- **1-m relationships**
 - The primary key from the ‘one side’ is placed as a foreign key in the ‘many side’.
- **m-n relationships**
 - A new relation is created with the primary keys from each entity forming a composite key.

• N-ary Relationships



A (a, m)

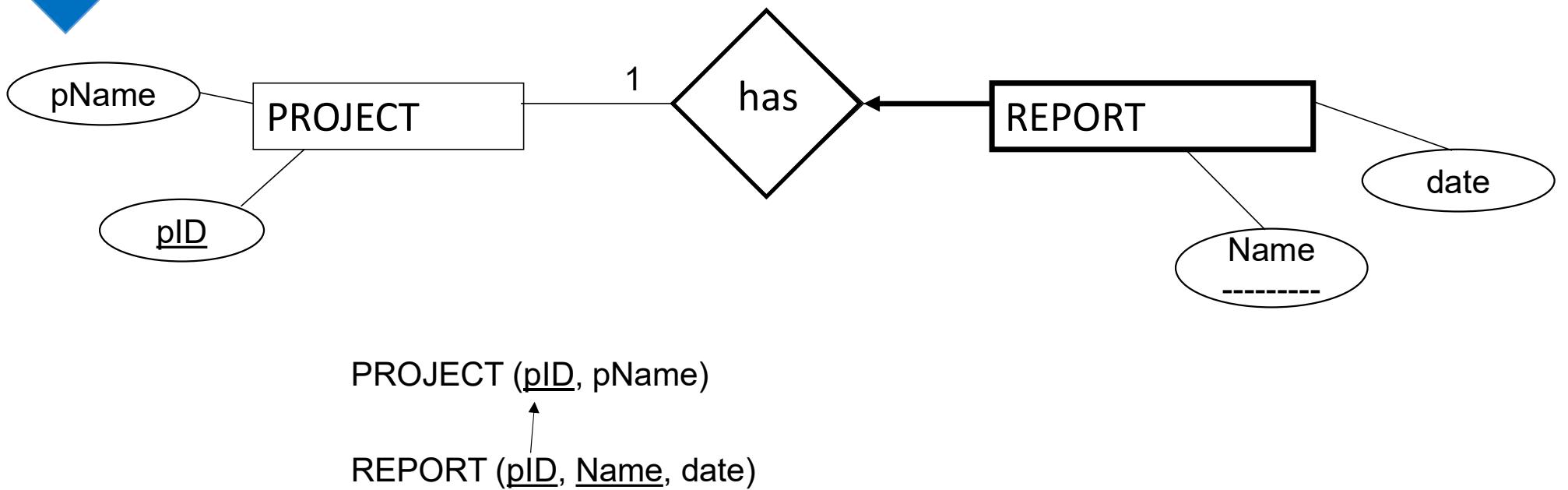
B (b, n)

C (c, p)

D (d, q)

R (a, b, c, d)

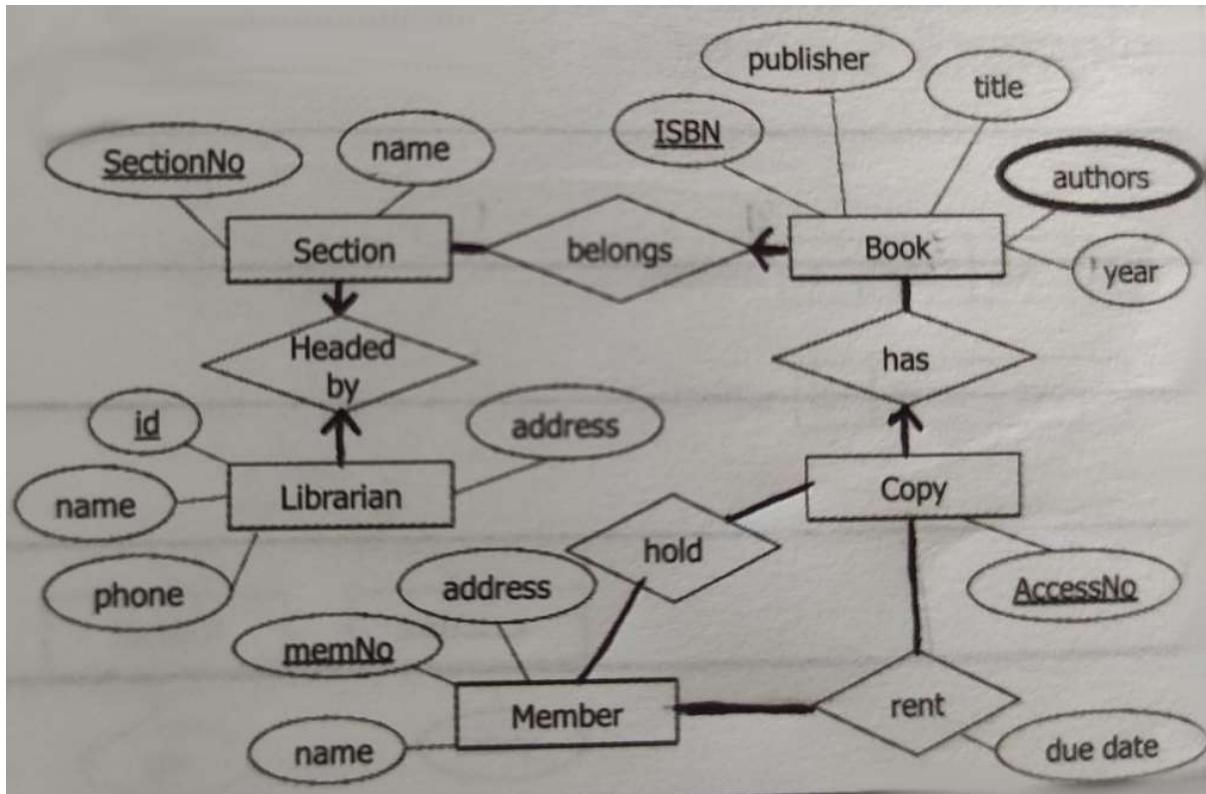
• Weak Entities



Activity 01

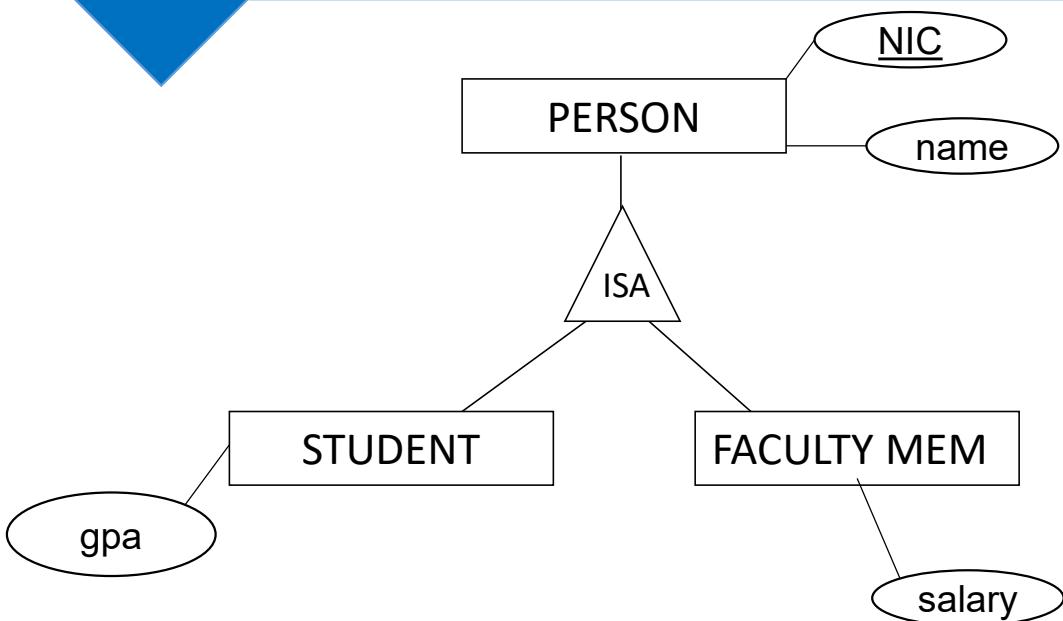


- Map the EER diagram to a relational schema.



Logical Mapping For EER Relationships

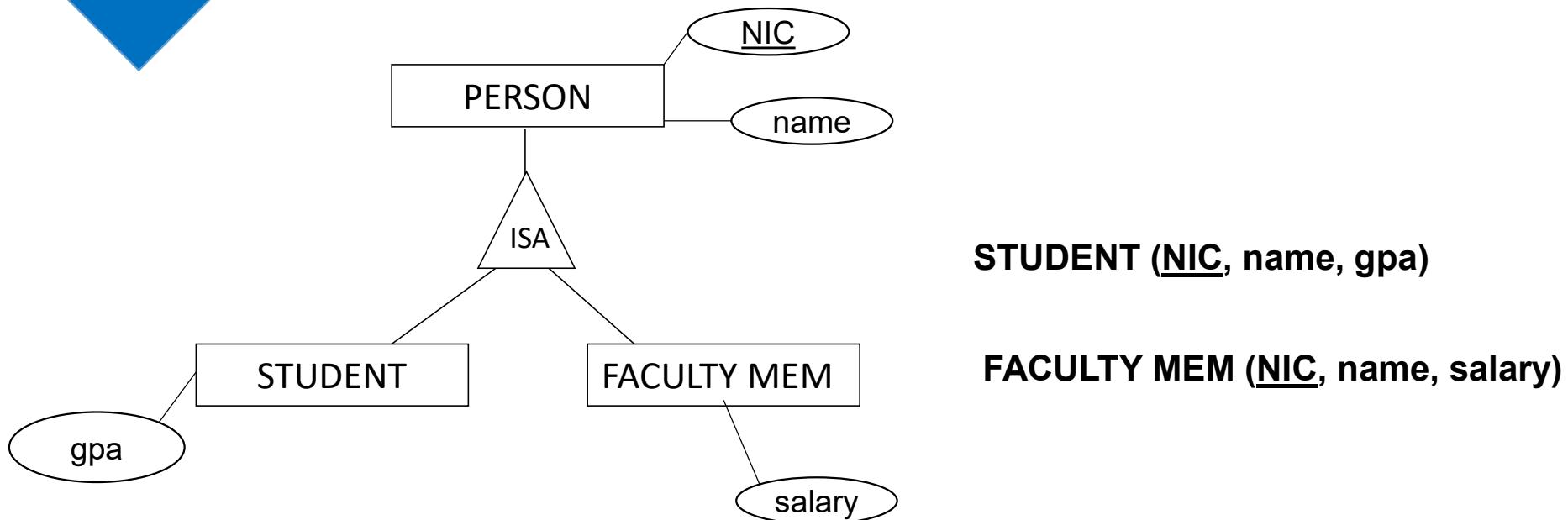
• ISA Relationship- Option 01



PERSON (NIC, name)
STUDENT (NIC, gpa)
FACULTY MEM (NIC, salary)

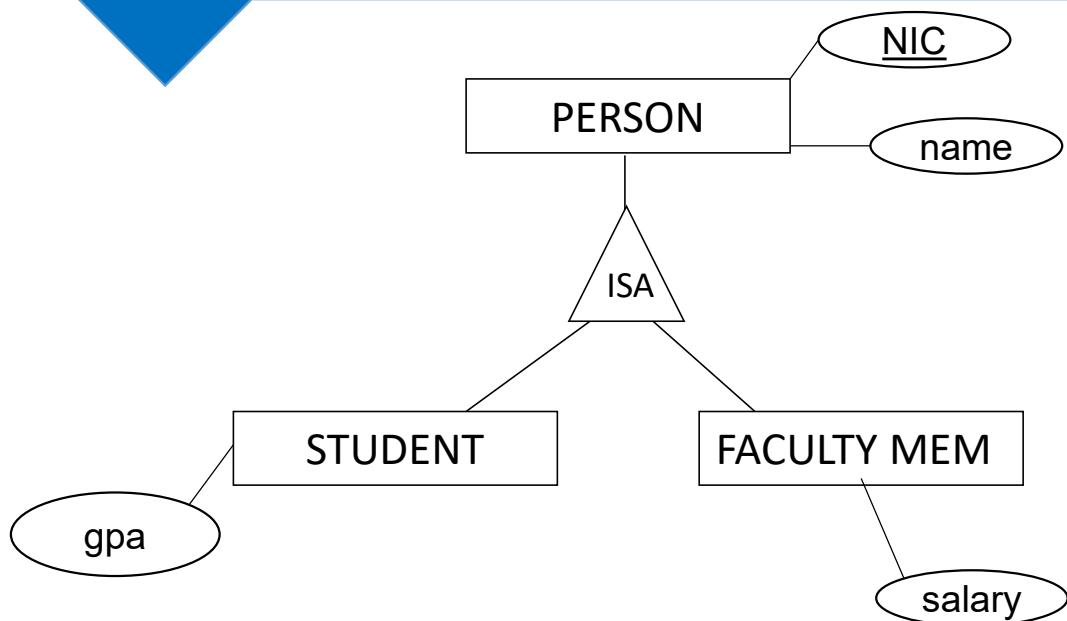
Option 01 works for all constraints

• ISA Relationship- Option 02



Option 02 works when there is a covering constraint

• ISA Relationship- Option 03

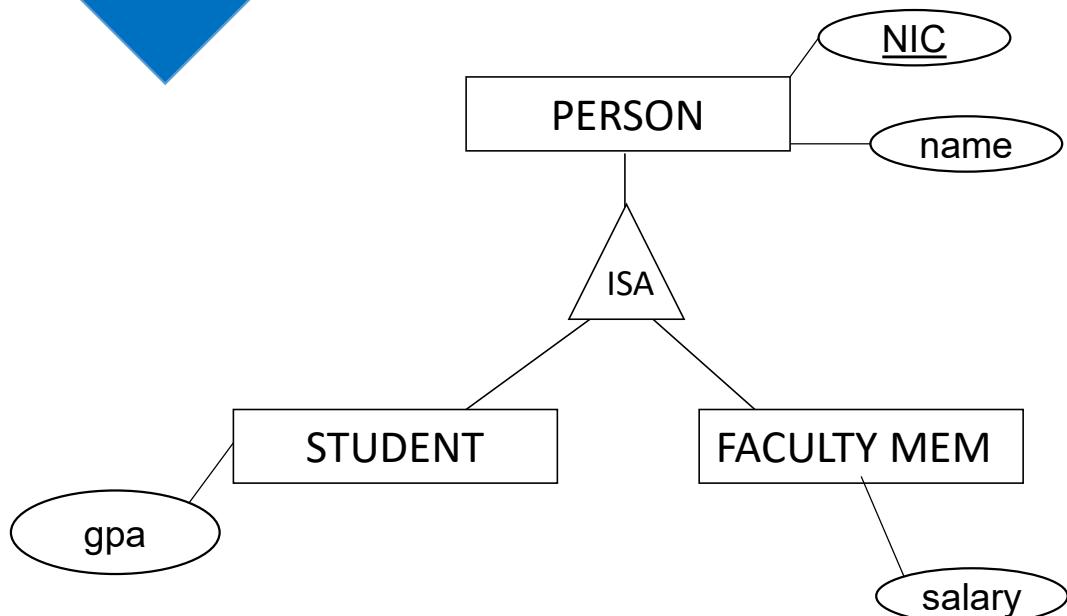


PERSON (NIC, name, salary, gpa, p_type)

Student / Faculty Mem

**Option 03 works for all disjoint
Good if subclasses have few attributes**

• ISA Relationship- Option 04



PERSON (NIC, name, salary, gpa, faculty_mem, student)

True / False

Option 04 works for all overlapping constraints

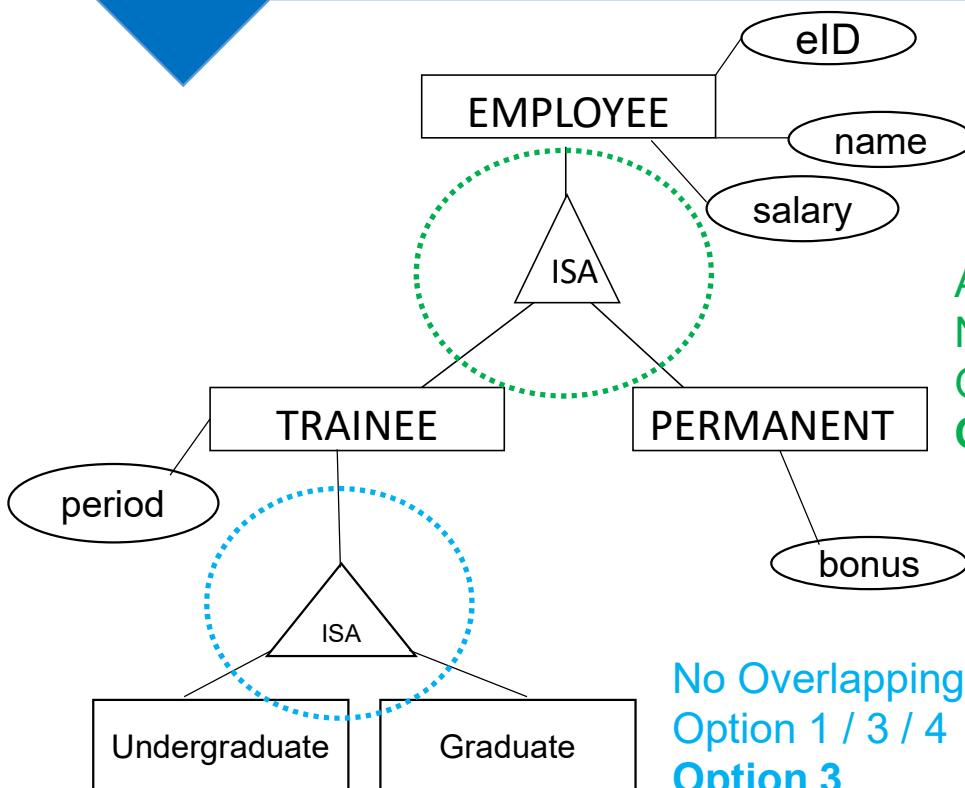
Activity 02



- Summarize the super class and sub class mapping

Option	Overlapping	Disjoint	Total	Partial
Option 1				
Option 2				
Option 3				
Option 4				

• ISA Relationship- Multilevel



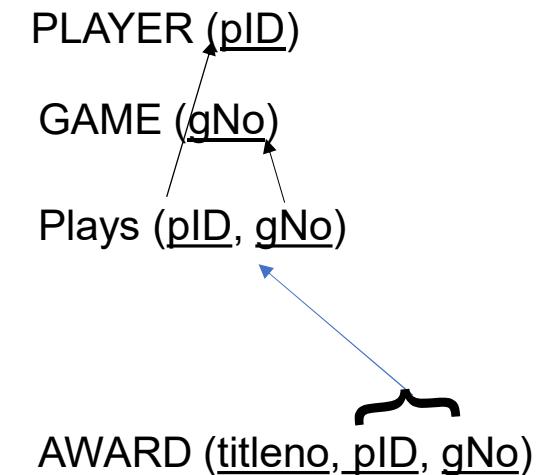
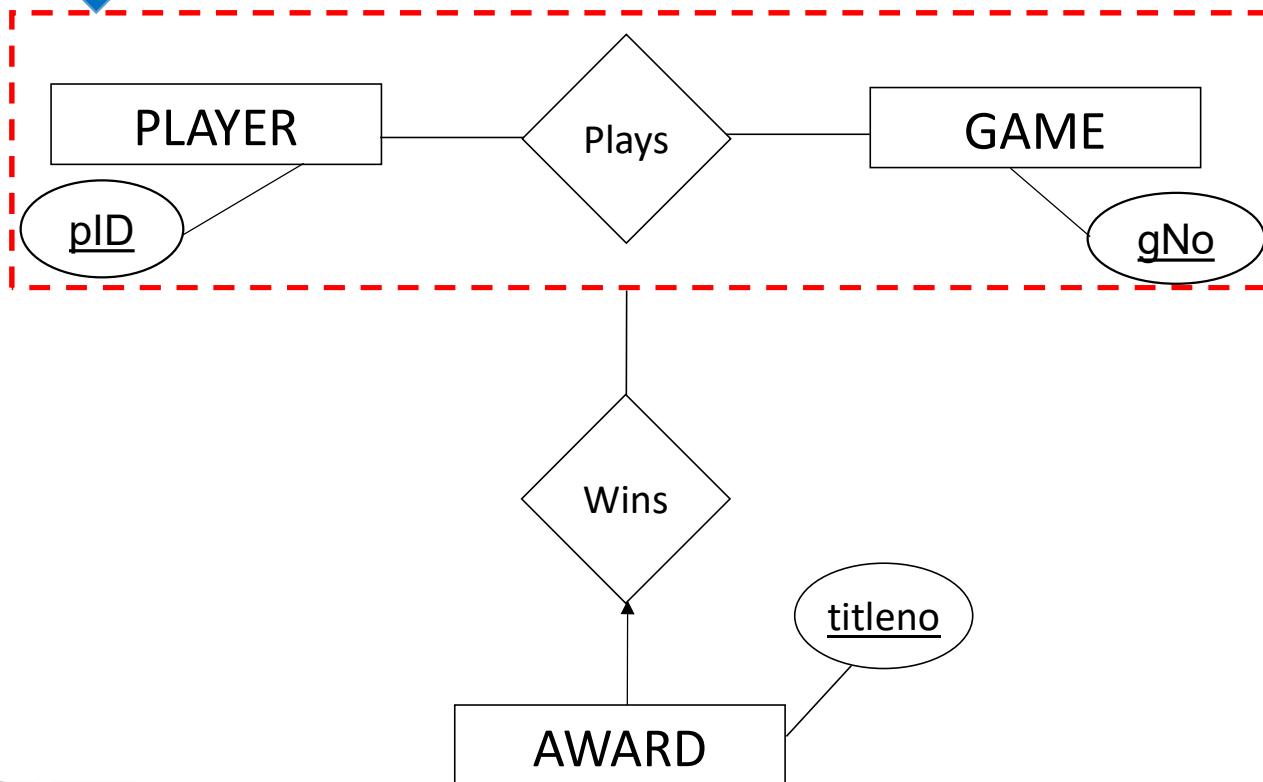
All employees are either trainee or permanent = Covering = Total
No overlapping = Disjoint
Option 1 / 2 / 3 / 4
Option 2

Trainee (`eid`, name, salary, period, type)
Permanent (`eid`, name, salary, bonus)

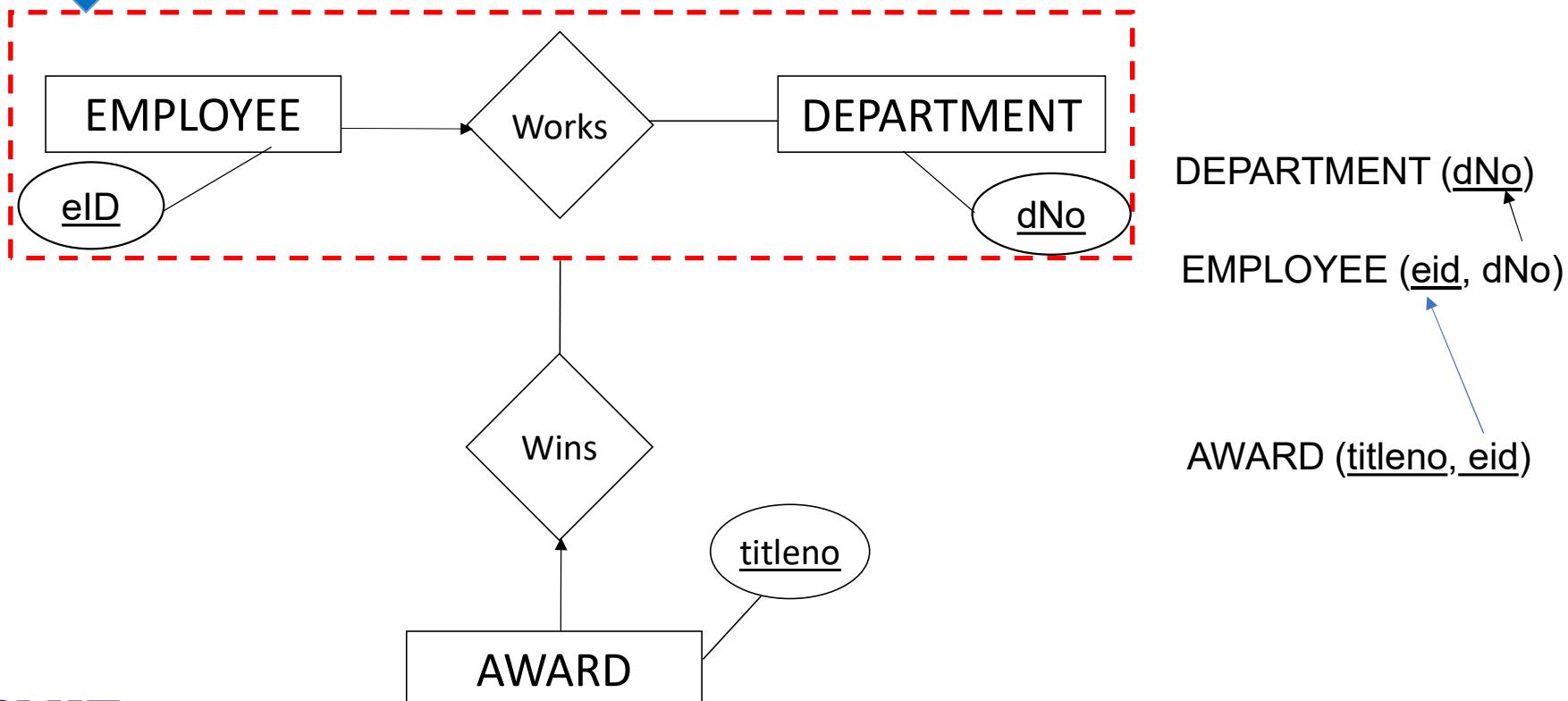
No Overlapping = Disjoint
Option 1 / 3 / 4
Option 3

Can use different options for different parts

• Aggregation

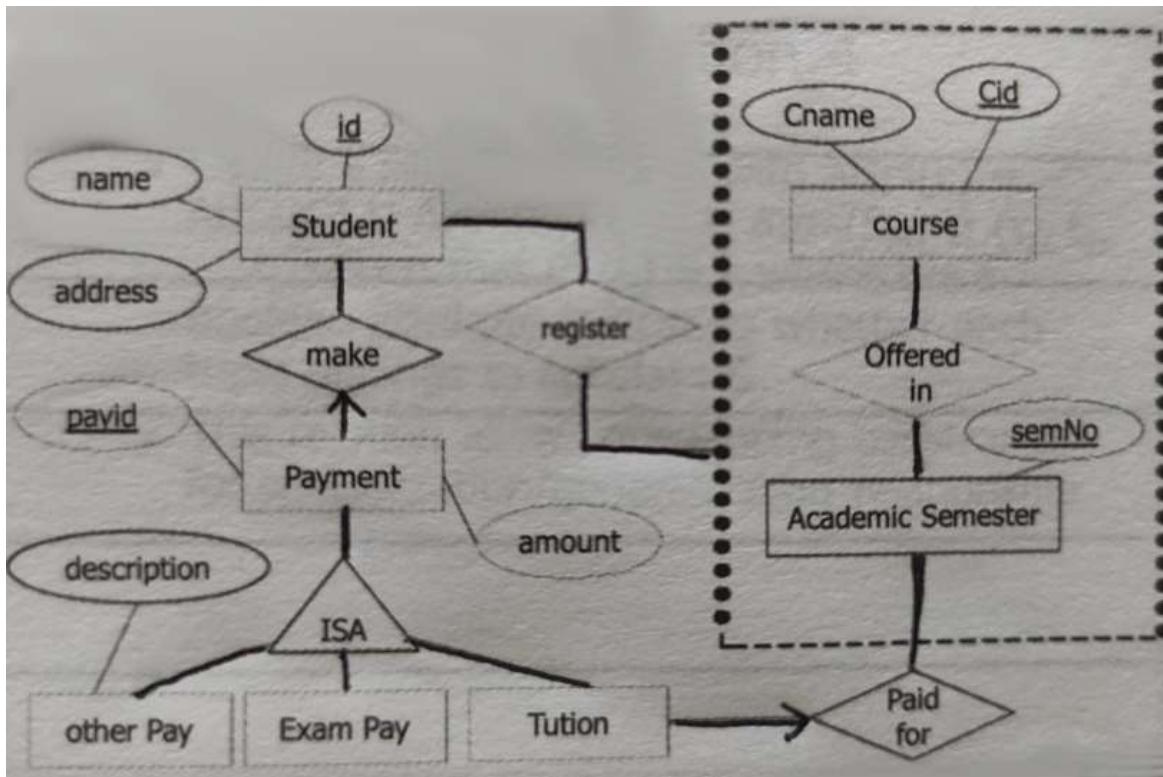


• Aggregation



Activity 03

- Map the EER diagram to a relational schema.





SLIIT

Discover Your Future

SCHEMA REFINEMENT



**COMPUTER
SYSTEMS
ENGINEERING**

IE2042 | DMSS | Lecture 3 | Dr. Harinda Fernando

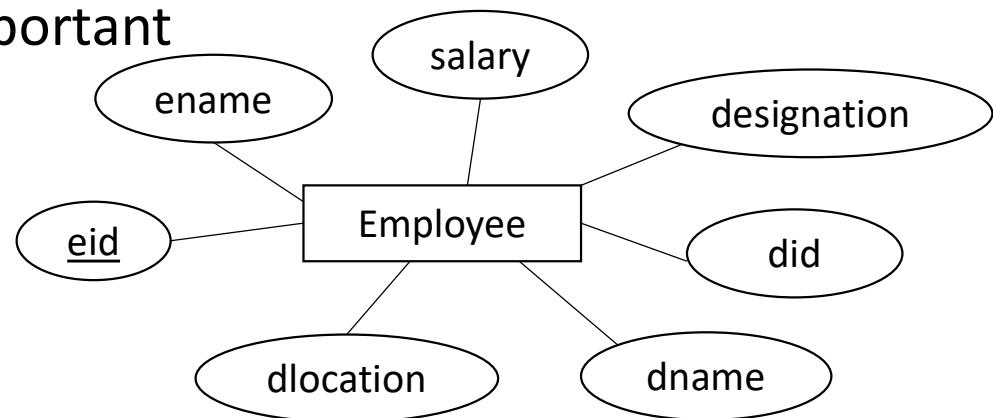
Learning Outcomes [L01]

- ▶ Understand the importance of schema refinement
- ▶ Explain the properties of good decomposition
- ▶ Understand functional dependencies
- ▶ Ability to compute keys from a set of functional dependencies in a relation
- ▶ Apply normal forms



Schema Refinement

- The relations resulted through logical DB design may be good or bad
- How we group attributes are very important
- If there are too many attributes;
 - **Waste Space**
 - **Anomalies**



Employee	<u>eid</u>	ename	designation	salary	did	dname	location
----------	------------	-------	-------------	--------	-----	-------	----------

Why Schema Refinement?

- **Insertion Anomaly**

- Inserting a new employee to the **emp** table
 - Department information is repeated (ensure that correct department information is inserted)
- Inserting a department with no employees
 - Impossible since **eid** cannot be null

- **Deletion Anomaly**

- Deleting the last employee from the department will lead to loosing information about the department

- **Update Anomaly**

- Updating the department's location needs to be done for all employees working for that department

Functional Dependency

- Functional dependency is a relationship that exists when one attribute uniquely determines another attribute
- For example: Suppose we have a student table with attributes: id, name, age
 - **id** attribute uniquely identifies the **name** attribute of student table, because if we know the student id we can tell the student name associated
 - This is known as functional dependency and can be written as **id->name** or in words we can say **name is functionally dependent on id** or **name is functionally determined by id**
- Redundancies in relations are based on functional dependencies
- Normalization is based on keys and functional dependencies

Types of Functional Dependencies

- Trivial ($X, Y \rightarrow X$)
- Semi-Trivial ($X, Y \rightarrow Y Z$)
- Non-Trivial ($X \rightarrow Z$)
- Multi-Valued ($X \rightarrow Y, X \rightarrow Z$)
- Transitive ($X \rightarrow Y, Y \rightarrow Z$)

Decomposition

<u>eid</u>	ename	designation	salary	did
1000	Ajith	Lecturer	60000	1
1001	Sunil	Executive	45000	3
1002	Kamal	Lecturer	75000	1
1003	Piyumi	Manager	50000	2
1004	Roshan	Lecturer	35000	1
1005	Nuwan	Lecturer	80000	1
1006	Jayamini	Assistant	25000	2
1007	Nishani	Lecturer	42000	1
1008	Amal	Assistant	28000	4

did	dname	location
1	Academic	Malabe
2	Admin	Metro
3	Maintenance	Kandy
4	ITSD	Matara

- Random decompositions may introduce new problems
- Consider the following properties to overcome the problems;
 - **Loss-Less Join Property**
 - **Dependency Preserving Property**

Why Schema Refinement?

<u>eid</u>	ename	designation	salary	did	dname	location
1000	Ajith	Lecturer	60000	1	Academic	Malabe
1001	Sunil	Executive	45000	3	Maintenance	Kandy
1002	Kamal	Lecturer	75000	1	Academic	Malabe
1003	Piyumi	Manager	50000	2	Admin	Metro
1004	Roshan	Lecturer	35000	1	Academic	Malabe
1005	Nuwan	Lecturer	80000	1	Academic	Malabe
1006	Jayamini	Assistant	25000	2	Admin	Metro
1007	Nishani	Lecturer	42000	1	Academic	Malabe
1008	Amal	Assistant	28000	4	ITSD	Matara

- Any duplicates?
- Any issues during insert / update / delete?

Loss Less Join Property

- Ability to recover the original relation from the decomposed relations

S	P	D
S1	P1	D1
S2	P2	D2
S3	P1	D3

S	P
S1	P1
S2	P2
S3	P1

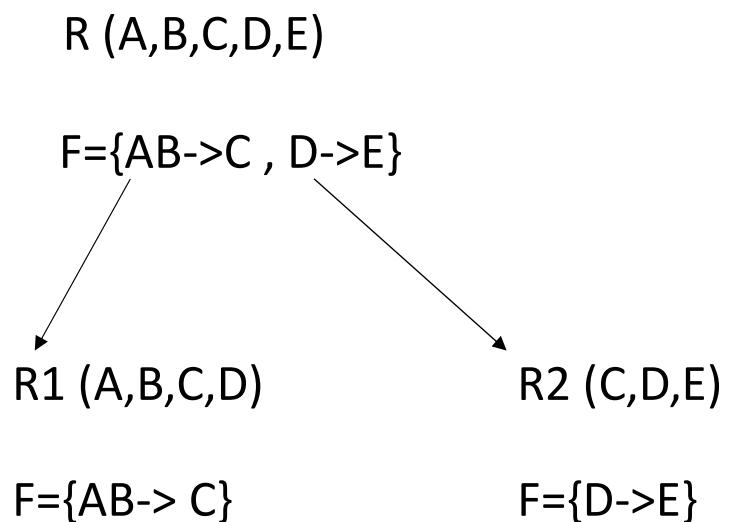
P	D
P1	D1
P2	D2
P1	D3

R1	R2
S1	P1
S1	P1
S2	P2
S3	P1
S3	P1

Spurious Tuples

Dependency Preserving Property

- Ability to enforce the constraints on the original relation, simply by enforcing constraints on each decomposed relation



Activity 01



- Consider the Employee Entity and write three functional dependencies that can exist in the Employee table.

Armstrong Axioms

- Can be used to identify all functional dependencies that exist on a relation
- 7 rules exist
 - 3 primary
 - 6 secondary – proven using the primary rules
- Notation

Let $R(U)$ be a relation scheme over the set of attributes U . Henceforth we will denote by letters X, Y, Z any subset of U and, for short, the union of two sets of attributes X and Y by XY instead of the usual $X \cup Y$; this notation is rather standard in [database theory](#) when dealing with sets of attributes.

Armstrong Axioms

- **Reflexivity**

If $X \subseteq Y$, then $Y \rightarrow X$

- **Augmentation**

If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z

- **Transitivity**

If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

eid	ename	designation	salary	did	dname	location
1000	Ajith	Lecturer	60000	1	Academic	Malabe
1001	Sunil	Executive	45000	3	Maintenance	Kandy
1002	Kamal	Lecturer	75000	1	Academic	Malabe
1003	Piyumi	Manager	50000	2	Admin	Metro
1004	Roshan	Lecturer	35000	1	Academic	Malabe
1005	Nuwan	Lecturer	80000	1	Academic	Malabe
1006	Jayamini	Assistant	25000	2	Admin	Metro
1007	Nishani	Lecturer	42000	1	Academic	Malabe
1008	Amal	Assistant	28000	4	ITSD	Matara

Armstrong Axioms

- **Union**

If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$

- **Composition**

If $X \rightarrow Y$ and $A \rightarrow B$, then $XA \rightarrow YB$

- **Decomposition**

If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$

- **Pseudo Transitivity**

If $X \rightarrow Y$ and $YZ \rightarrow W$ then $XZ \rightarrow W$

<u>eid</u>	<u>ename</u>	<u>designation</u>	<u>salary</u>	<u>did</u>	<u>dname</u>	<u>location</u>
1000	Ajith	Lecturer	60000	1	Academic	Malabe
1001	Sunil	Executive	45000	3	Maintenance	Kandy
1002	Kamal	Lecturer	75000	1	Academic	Malabe
1003	Piyumi	Manager	50000	2	Admin	Metro
1004	Roshan	Lecturer	35000	1	Academic	Malabe
1005	Nuwan	Lecturer	80000	1	Academic	Malabe
1006	Jayamini	Assistant	25000	2	Admin	Metro
1007	Nishani	Lecturer	42000	1	Academic	Malabe
1008	Amal	Assistant	28000	4	ITSD	Matara

Attribute Closure

- Set of attributes that could be determined by an attribute
- Closure is obtained by repeatedly using the axioms
- Denoted by **[X]+**
- If **X+ = All attributes**, then X is a **CANDIDATE KEY**

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

Closure of A; $A^+ = \{A, B, C\}$

Attribute Closure

IF R (A,B,C,D, E) and FD { $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow D$, $D \rightarrow E$ }

- Find closure of A and BC
 - Consider A
 - $A \rightarrow A$ (Reflexivity)
 - $A \rightarrow B$ (Given)
 - $A \rightarrow C$ (Transitivity)
 - $A \rightarrow D$ (Transitivity)
 - $A \rightarrow E$ (Transitivity)
 - Therefore $A^+ = \{A, B, C, D, E\}$

Keys - Revision

- Super Key – set of attributes which uniquely identify all attributes
- Candidate Key – Super key which cannot have any columns removed
- Primary Key – candidate key chosen to be the main key
- Important for normalization

Activity 2.1



- Consider the relation **R (A,B,C,D)** with the following functional dependencies.

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

What are the keys of this relation?

- Use Armstrong Axioms
- Without using the axioms

Activity 2.2



- Consider the relation **R (A,B,C,D,E)** with the following functional dependencies.

$$F = \{A \rightarrow B, A \rightarrow C, CD \rightarrow E, B \rightarrow D, E \rightarrow A\}$$

What are the keys of this relation?

- Use Armstrong Axioms
- Without using the axioms

Normal Forms

- A series of tests performed on relational schemas to reduce problems
- The four main;
 - **1st Normal Form**
 - **2nd Normal Form**
 - **3rd Normal Form**
 - **Boyce-Codd Normal Form**
- The normal form of a relation is the highest normal form condition it meets
- Top down approach

1st Normal Form

- A relation R is in first normal form (1NF) if domains of all attributes in the relation are **atomic** (simple & indivisible)

Remove Multi-value attributes

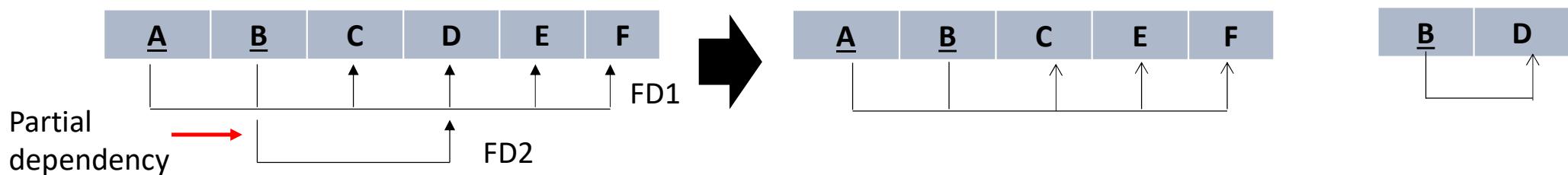
Remove Composite attributes

2nd Normal Form

- A relation R is in second normal form (2NF) if every nonprime attribute A in R is not partially dependent on any key of R.

Only Full Functional Dependency allowed

Remove any Partial Dependencies



Activity 03



- What normal form the relation below? If the relation is not in 2NF normalize the relation to 2NF.

EMP_PROJ					
NIC	PNUM	HOURS	ENAME	PNAME	LOC
FD1					
FD2					
FD3					

FD1 | | | ↑

FD2 | | | ↑

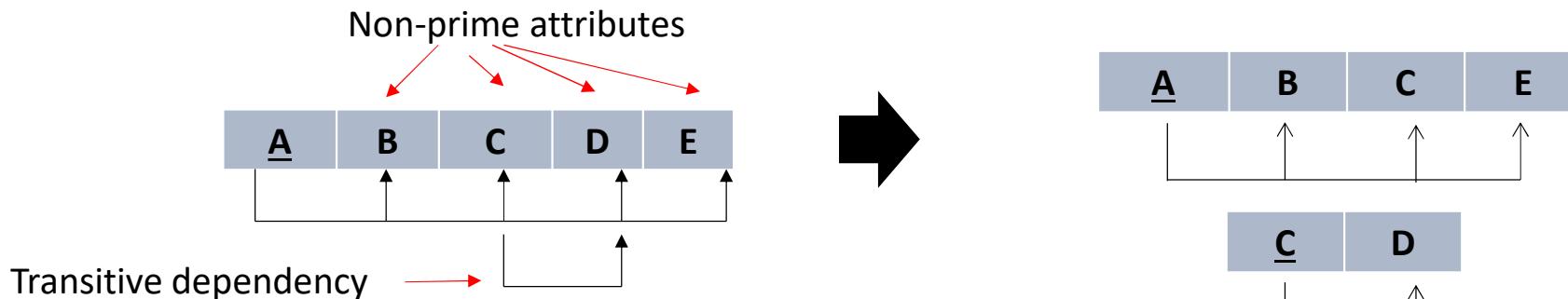
FD3 | | | ↑ | ↑

3rd Normal Form

- A relation R is in third normal form (3NF) if R is in 2NF and no non prime attribute is transitively dependent on any key

Prime -> Prime/Non Prime Allowed

Remove Non Prime- Non Prime Dependencies (Transitive)



Activity 04



- What normal form the relation below? If the relation is not in 3NF normalize the relation to 3NF.

EMP_DEPT

ENAME	<u>SSN</u>	BDATE	ADD	DNUM	DNAME	DMGR

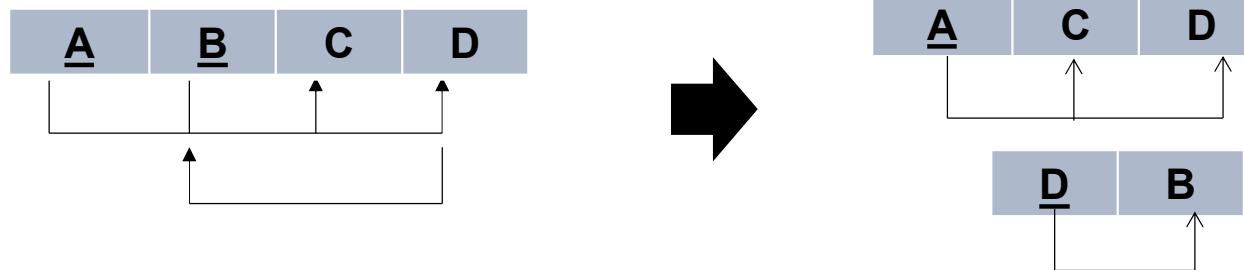
The diagram illustrates the primary key dependencies for the EMP_DEPT relation. The primary key is underlined in the header: SSN. Arrows point from the other attributes to the primary key: ENAME points to SSN, BDATE points to SSN, ADD points to SSN, DNUM points to SSN, DNAME points to SSN, and DMGR points to SSN.

Boyce-Codd Normal Form

- A relation R is in Boyce Codd normal form if R is in 3NF and if in every nontrivial functional dependency $X \rightarrow Y$, X is a Super Key

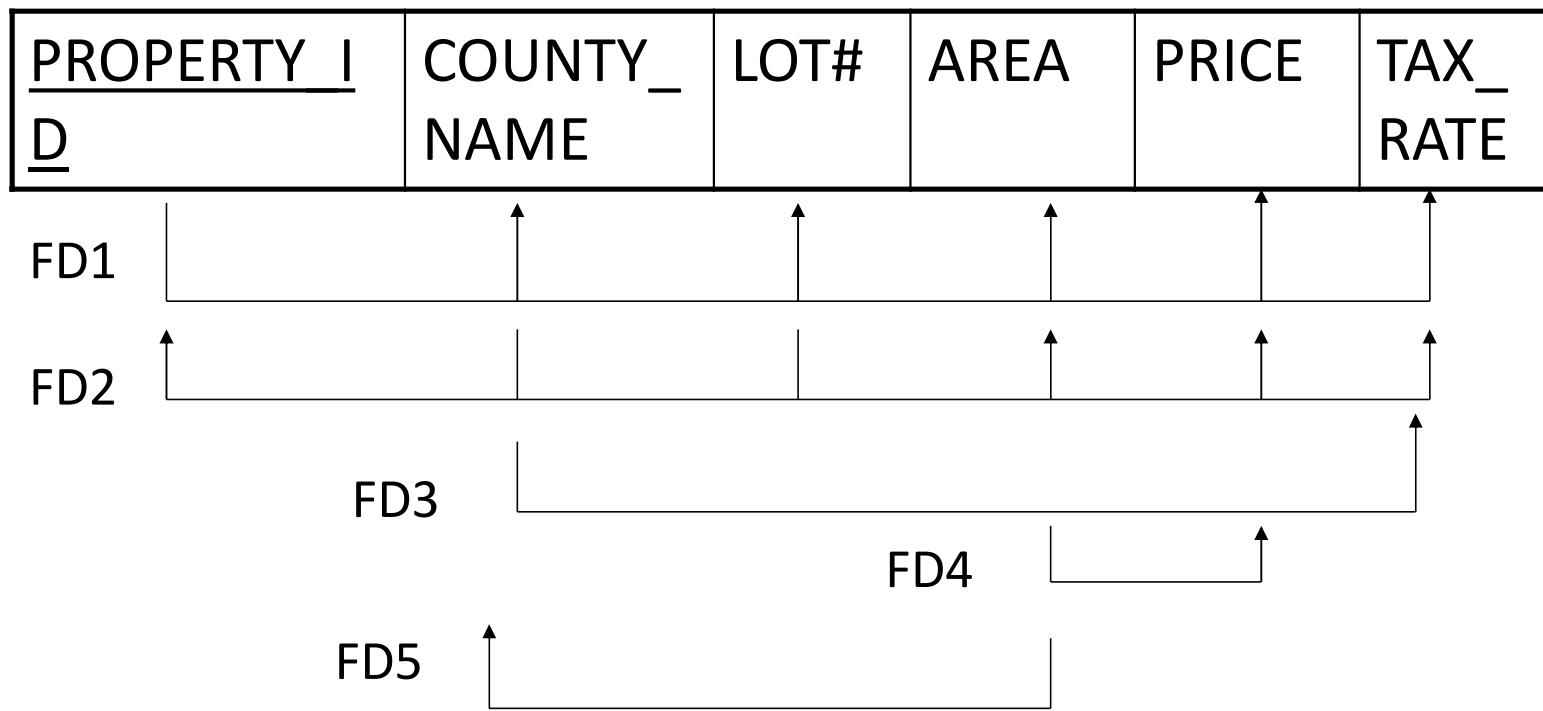
Prime \rightarrow Prime/Non Prime Allowed

Remove Non Prime- Prime Dependencies



Activity 05

- Normalize the relation below.

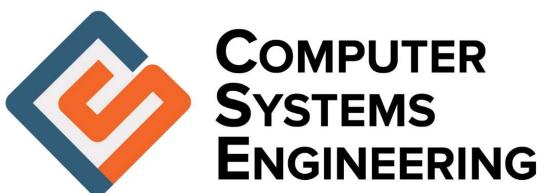




SLIIT

Discover Your Future

SQL



IE2042 | DMSS | Lecture 5 | Dr. Harinda Fernando

Learning Outcomes (L01, L02)

- ▶ Understand the data definition language in SQL
- ▶ Understand the data manipulation language in SQL
- ▶ Write syntactically correct SQL statements to retrieve data



History of SQL

- Initially called SEQUEL (for Structured English QUERy Language)
- It was developed for an experimental relational database system called System R
- A joint effort between ANSI (American National Standard Institute) and ISO (International Standards Organization) led to a standard version of SQL in 1986 (SQL1, SQL-86, etc.)
- Major revisions have been proposed and SQL2 (also called SQL-92) has subsequently been developed

Relational Model vs SQL

Relational Model	SQL
Relation	Table
Attribute	Column
Tuple	Row

SQL

SQL is a comprehensive database language:

- Data Definition Language (DDL)
- Data Manipulation Language (DML)
- Facilitates for security & authorization
- Facilitates for transaction processing
- Facilitates for embedding SQL in general purpose languages (Embedded SQL)

Data Definition Language (DDL)

- DDL is the subset of SQL that supports the creation, deletion and modifications for tables and views
- Constraints can be defined on the tables

Constraint	Purpose
Not Null	Ensure that column doesn't have null values
Unique	Ensure that column doesn't have duplicate values
Primary key	Defines the primary key
Foreign key	Defines a foreign key
Default	Defines a default value for a column (When no values are given)
Check	Validates data in a column

DDL- Create Table

```
CREATE TABLE STUDENT
```

```
(
```

```
studentId INTEGER PRIMARY KEY,
```

```
sName VARCHAR (30) NOT NULL,
```

```
nic CHAR(10) UNIQUE,
```

```
gpa FLOAT,
```

```
progId VARCHAR(10) DEFAULT 'IT',
```

```
CONSTRAINT student_prog_fk FOREIGN KEY (progId) REFERENCES programs(id),
```

```
CONSTRAINT gpa_ck CHECK (gpa<= 4.0 )
```

```
)
```

DDL- Modify / Delete Table

- ALTER command – to alter the definition of the object
 - Ex : Adding a new column to a table
 - **ALTER TABLE STUDENT ADD age INT**
 - Ex : Adding a new constraint to a column
 - **ALTER TABLE STUDENT ADD CONSTRAINT chk_age CHECK (age > 18)**
 - Ex : removing a column from a table
 - **ALTER TABLE STUDENT DROP COLUMN age**
- DROP command – to drop objects
 - Ex: Deleting a table
 - **DROP TABLE STUDENT**

Data Manipulation Language (DML)

- DML is the subset of SQL that allow users to write statements to insert, delete, modify and display rows
 - Inserting a row
 - **INSERT INTO STUDENT VALUES (1000, 'Amal', '923456789V', 3.2, 'BM')**
 - **INSERT INTO STUDENT(studentId, sName, nic) VALUES (1001, 'Nimali', '834567890V')**

studentID	sName	nic	gpa	progId
1000	Amal	923456789V	3.2	BM
1001	Nimali	834567890V	Null	IT

Data Manipulation Language (DML)

- Deleting a row
 - **DELETE STUDENT WHERE** studentId=1000
- Updating a row
 - **UPDATE STUDENT**
SET gpa=2.8
WHERE studentId=1001

Select Clause

- Select clause in SQL is the basic statement for retrieving information from a database

```
SELECT <attributes>  
FROM   <one or more relations>  
WHERE <conditions>
```

Ex : Display ids of all students whose gpa is above 3.0

```
Select studentId  
From STUDENT  
Where gpa> 3.0
```

Clauses and Operators used with SELECT

- LIKE operator
- IS [NOT] NULL operator
- DISTINCT operators
- BETWEEN operator
- ORDER BY clause
- Joins (inner & outer)
- Nested query (IN/SOME/ANY, ALL), [NOT] EXISTS
- Aggregate functions
- GROUP BY – HAVING clauses

LIKE Operator

- Used for matching patterns
- Syntax : <string> LIKE <pattern>
 - <pattern> may contain two special symbols:
 - % = any sequence of characters
 - _ = any single character

Ex : Find students whose name start with an 'A'

Select Name

From Student

Where Name Like 'A%'

Student

StudentID	Name	gpa	progid
1000	Amal	3.2	BM
1001	Nimali	Null	IT
1002	Aruni	3.0	SE
1003	Surani	2.5	IT

Name
Amal
Aruni

IS [NOT] NULL Operator

- Used to check whether attribute value is null

Ex : Find StudentIDs of the students who have not completed a semester yet.

Select StudentID

From Student

Where gpa IS NULL



StudentID
1001
1004

Student

StudentID	Name	gpa	progId
1000	Amal	3.2	BM
1001	Nimali	Null	IT
1002	Aruni	3.0	SE
1003	Surani	2.5	IT
1004	Imali	Null	BM

DISTINCT Operator

- In a table, a column may contain many duplicate values
- Duplicates in results can be eliminated using DISTINCT operator

Ex :

Select progl
From Student

Progl
BM
IT
SE
IT
BM

Select DISTINCT progl
From Student

Student			
StudentID	Name	gpa	progl
1000	Amal	3.2	BM
1001	Nimali	Null	IT
1002	Aruni	3.0	SE
1003	Surani	2.5	IT
1004	Imali	Null	BM

Progl
BM
IT
SE

BETWEEN Operator

- Used to check whether attribute value is within a range

Ex : Find the students who will be obtaining a first class ($3.7 \leq \text{gpa} \leq 4.0$)

```
Select StudentID  
From Student  
Where gpa between 3.7 and 4.00
```

Student			
StudentID	Name	gpa	progId
1000	Amal	3.2	BM
1001	Nimali	Null	IT
1002	Aruni	3.8	SE
1003	Surani	2.5	IT
1004	Imali	4.0	BM



StudentID
1002
1004

ORDER BY Clause

- Used to order results based on a given field
- Ordering is ascending (ASC), unless you specify the DESC keyword

Ex : Display the student names and gpa's in the ascending order of gpa's.

Select Name, gpa
From Student
Order by gpa



Name	gpa
Surani	2.5
Nimali	2.8
Amal	3.2
Aruni	3.8
Imali	4.0

Student

StudentID	Name	gpa	progId
1000	Amal	3.2	BM
1001	Nimali	2.8	IT
1002	Aruni	3.8	SE
1003	Surani	2.5	IT
1004	Imali	4.0	BM

[INNER] Join

- Join two tables based on a certain condition

Ex : Find the names of students who follow programs

offered by SLIIT

Select s.Name

From Student s, Program p

Where s.pid=p.progId and offerBy='SLIIT'

Or

Select s.Name

From Student s INNER JOIN Program p on s.pid= p.progId

Where offerBy='SLIIT'

Student

SID	Name	gpa	pid
1000	Amal	3.2	BM
1001	Nimali	2.8	IT
1002	Aruni	3.8	SE
1003	Surani	2.5	IT

Program

progId	years	offerBy
BM	3	Curtin
IT	4	SLIIT
SE	3	SHU

Name
Nimali
Surani

LEFT OUTER Join

- Return all rows from the table on the left hand side of the join, with the matching rows in the table on the right hand side of the join
- The result is NULL in the right side when there is no match

Ex : For all the students display the name and the offering institute

Select s.Name, p.offerBy

From Student s LEFT OUTER JOIN Program p on s.pid=p.progId

Student

SID	Name	gpa	pid
1000	Amal	3.2	BM
1001	Nimali	2.8	IT
1002	Aruni	3.8	SE
1003	Surani	2.5	IT

Program

progId	years	Offer By
BM	3	Curtin
IT	4	SLIIT

Name	offerBy
Amal	Curtin
Nimali	SLIIT
Aruni	NULL
Surani	SLIIT

RIGHT OUTER Join

- Return all rows from the table on the right hand side of join, with the matching rows in the table on the left hand side of the join
- The result is NULL in the left side when there is no match

Ex : For all the programs display the offering institute and names of the students following it.

Select s.Name, p.offerBy

From Student s RIGHT OUTER JOIN Program p on s.pid=p.progId

Student

SID	Name	gpa	pid
1000	Amal	3.2	BM
1001	Nimali	2.8	IT
1002	Aruni	3.8	SE
1003	Surani	2.5	IT

Program

progId	years	offerBy
BM	3	Curtin
IT	4	SLIIT
SE	3	SHU

Name	offerBy
Amal	Curtin
Nimali	SLIIT
Surani	SLIIT
NULL	SHU

IN Operator

- Used to check whether attribute value matches any value within a value list

Ex : Find the student names who have obtained an 'A'.

Select s.Name

From Student s

Where s.SID IN (Select SID

From Grades

Where Grade='A')

Name
Amal
Nimali

Student

SID	Name	gpa
1000	Amal	3.2
1001	Nimali	2.8
1002	Aruni	3.8

Grades

SID	cid	Grade
1000	IT102	A
1000	IT100	B
1001	IT102	A
1002	IT102	C
1002	IT200	C

EXISTS Operator

- Used to check if subquery returns any rows

Ex : Find the students who have obtained an 'A'.

Select s.Name

From Student s

Where EXISTS (Select *

From Grades g

Where g.SID=s.SID and g.Grade='A')

Student		
SID	Name	gpa
1000	Amal	3.2
1001	Nimali	2.8
1002	Aruni	3.8

Grades

SID	cid	Grade
1000	IT102	A
1000	IT100	B
1001	IT102	A
1002	IT102	C
1002	IT200	C

Name
Amal
Nimali

ANY and SOME Operators

- Compare a value to a list or subquery and return true if at least one of the comparison evaluates as true

Ex : Find BM students whose gpa is greater than any of the IT students.

```
Select s.Name  
From Student s  
Where s.progID='BM' and s.gpa > ANY
```

```
(select s1.gpa  
From student s1  
Where s1.progID='IT')
```

Student			
StudentID	Name	gpa	progId
1000	Amal	2.7	BM
1001	Nimali	2.8	IT
1002	Aruni	3.8	SE
1003	Surani	2.5	IT
1004	Imali	4.0	BM

Name
Amal
Imali

ALL Operator

- If all of the comparisons evaluate to true then the result of the ALL expression will be true. Otherwise the result will be false

Ex : Find BM students whose gpa is greater than all the IT students.

```
Select s.Name  
From Student s  
Where s.progID='BM' and s.gpa > ALL
```

```
(Select s1.gpa  
From Student s1  
Where s1.progId='IT')
```

Student			
StudentID	Name	gpa	progId
1000	Amal	2.7	BM
1001	Nimali	2.8	IT
1002	Aruni	3.8	SE
1003	Surani	2.5	IT
1004	Imali	4.0	BM

Name
Imali

Aggregation

- Summarize the results of an expression over a number of rows by returning a single row
- Commonly used- SUM, COUNT, AVG, MIN and MAX

Ex : Find average, minimum and the maximum gpa.

Select AVG(gpa), MIN(gpa), MAX(gpa)

From Student

Student			
StudentID	Name	gpa	progId
1000	Amal	3.2	BM
1001	Nimali	2.8	IT
1002	Aruni	3.8	SE
1003	Surani	2.5	IT
1004	Imali	4.0	BM

AVG(gpa)	MIN(gpa)	MAX(gpa)
3.26	2.5	4.0

Grouping (GROUP BY Clause)

- Groups the data in tables and produces a single summary row for each group
- Grouping is done based on values in a given field
- When using group by
 - Each item in the SELECT list must be single valued per group.
 - SELECT clause may only contain Columns names, aggregate function, constants or an expression involving combinations of the above.
 - All column names in SELECT list must appear in the GROUP BY clause unless the name is used only in the aggregate function.

GROUP BY

Ex : Count the number of students
who has followed each module.

Select CID, Count(SID)
From Student
Group by CID

Student		
SID	CID	Grade
1000	DBII	A
1000	SEI	B
1001	DBII	A
1002	DBII	C
1002	SPD	C

CID	Count (SID)
DBII	3
SEI	1
SPD	1

SID	CID	Grade
1000	DBII	A
1001	DBII	A
1002	DBII	C
1000	SEI	B
1002	SPD	C

HAVING Clause

- Used to apply conditions on the groupings

Ex : Find courses followed by more than two students

```
Select CID, Count(SID)  
From Student  
Group by CID  
Having count(SID)>2
```

Student		
SID	CID	Grade
1000	DBII	A
1000	SEI	B
1001	DBII	A
1002	DBII	C
1002	SPD	C

SID	CID	Grade
1000	DBII	A
1001	DBII	A
1002	DBII	C
1000	SEI	B
1002	SPD	C

CID	Count (SID)
DBII	3

Summary

SELECT <attribute list>

FROM <table list>

WHERE <condition>

GROUP BY <group attribute(s)>

HAVING <group condition>

ORDER BY <attribute list>



SLIIT

Discover Your Future

DATABASE PROGRAMMING



**COMPUTER
SYSTEMS
ENGINEERING**

IE2042 | DMSS | Lecture 7 | Dr. Harinda Fernando

Learning Outcomes (L01, L02)

- ▶ Understand what is a view and how to use it
- ▶ Understand situations where T-SQL is applicable
- ▶ Understand stored functions and procedures
- ▶ Understand situations where triggers can be used
- ▶ Write syntactically correct SQL statements to create views, functions, procedures and triggers



Views

- A **view** is a virtual table derived from other tables which are called **base/defining tables**.
 - Typically the result set of a SQL statement
 - A view does not necessarily exist in a physical form.
 - A view contains rows and columns, just like a real table.
 - You can apply SQL statements and functions to a view and present the data as if the data were coming from one single table.

Creating a View

- Create a view-

```
CREATE VIEW <view name> AS
```

```
    SELECT <column name(s)>
```

```
        FROM <table name(s)>
```

```
        WHERE <condition>
```

Views (contd)

- Query a view-

```
SELECT <column name(s) in the view>
```

```
FROM <view name>
```

```
WHERE <condition>
```

- Delete a view-

```
DROP VIEW <view name>
```

Exercise

- Try using the prebuilt and populated database at [SQL CREATE VIEW, REPLACE VIEW, DROP VIEW Statements \(w3schools.com\)](#) to work with views
- Create a view that shows the product details along with the supplier name and category that product belongs to

Answer

```
CREATE VIEW [Product Details] AS  
SELECT ProductID, Productname, Unit, Price, SupplierName,  
CategoryName  
FROM Products P, Suppliers S, Categories C  
WHERE P.SupplierID = S.SupplierID      and P.CategoryID =  
C.CategoryID;
```

```
SELECT * FROM [Product Details]
```

Activity 01

- Consider the schema used in tutorial 04 (emp,works,dept)
1. Create a view named department information that contains the name of the department, budget and the manager's name.
 2. Create a view named employee information that contains the employee id, name, salary and total percentage time.
 3. Select the manager who manages the highest budget.

Views (contd)

- Update a view-

UPDATE <view name>

SET <column name= new value>

WHERE <condition>

- Updating a view can be ambiguous
 - Views containing aggregate functions are not updatable
 - Views containing joins can be ambiguous
- Views are updatable when they are defined on a single base table

Views (contd)

- Views containing a join can be ambiguous

A	B
b	1

B	C
1	d
1	e

A	B	C
b	1	d
b	1	e

V1

UPDATE V1
SET A = a
WHERE C = e

Views (contd)

- Advantages
 - Security - Setting access levels or permission
 - Query Simplicity - Multi table queries can be turned to one view
- Disadvantages
 - Performance - Actually have to access base tables
 - Update restrictions - On aggregates and joins

T-SQL

- SQL is a structure query language which is a common database language for all RDBMS products.
- Different RDBMS product vendors have developed their own database language by extending SQL
- T-SQL stands for Transact Structure Query Language which is a Microsoft product and is an extension of SQL Language.
- Allows simple programming like commands to be used along with standard SQL commands

T-SQL Vs SQL

T-SQL

- While T-SQL is an extension to SQL (all SQL commands are supported)
- T-SQL contains procedural programming and local Variables
- T-SQL is proprietary

SQL

- SQL is a programming language.
- SQL does not
- SQL is an open format.

Programming in T-SQL

- Extensions have been made in SQL to program simple server-side logic.
- Some of the statements include:
 - **Variables**
 - **Selection conditions**
 - IF (...)..... ELSE ...
 - **Looping**
 - WHILE (...)

T-SQL- Variables

- A Transact-SQL local variable is an object that can hold a single data value of a specific type.
- Variables in scripts are typically used:
 - As a counter either to count the number of times a loop is performed or to control how many times a loop is performed
 - To hold a data value to be tested by a control-of-flow statement
 - To save a data value to be returned by a stored procedure return code

T-SQL- Variables

- In total T-SQL defines 7 different data types for use
 - Exact Numeric Types
 - Approximate Numeric Types
 - Date and Time Types
 - Character Strings
 - Unicode Character Strings
 - Other Data Types
- For full descriptions and subtypes refer to [T-SQL - Data Types
\(tutorialspoint.com\)](http://tutorialspoint.com)

T-SQL- Variables

- The **DECLARE** statement initializes a T-SQL variable
 - Syntax: **DECLARE @<variable name> <data type>**
 - Ex: `DECLARE @DName VARCHAR(20)`
- The **SET** statement assigns a value to the variable
 - Syntax : **SET @ <variable name> = <value>**
 - `SET @DName = 'SESD'`

T-SQL- Variables

- The declared variables could be used in scripts

- `SELECT budget`

- `FROM Department`

- `WHERE dname = @DName`

- `DECLARE @emplId INT`

- `SELECT @emplId = MAX(eid)`

- `FROM Employee`

Flow Control in T-SQL

- Keywords for flow control in Transact-SQL include
 - BEGIN and END - mark a block of statements
 - GOTO – Jumps to a specific line of code
 - IF and ELSE – Allows selection
 - RETURN - Used to immediately return from a stored procedure or function
 - WAITFOR - Wait for a given amount of time, or until a particular time of day
 - WHILE – Allows iteration of commands
 - BREAK – Ends the enclosing while loop
 - CONTINUE – Starts next iteration of enclosing while loop

T-SQL- IF Statements

- Imposes conditions on the execution of a T-SQL statement

```
IF (SELECT count(eid) FROM Employee) < 100
```

```
BEGIN
```

```
    PRINT 'Inside the IF statement'
```

```
    PRINT 'There are less than 100 employees'
```

```
END
```

```
ELSE
```

```
    PRINT 'There are more than 100 employees!'
```

T-SQL- WHILE Statements

- Sets a condition for the repeated execution of an SQL statement or statement block.
- The statements are executed repeatedly as long as the specified condition is true.

```
WHILE @count<=100
BEGIN
    INSERT INTO Employee VALUES(@count,CONCAT('Employee',@count))
    SET @count=@count+1
END
```

FUNCTIONS & PROCEDURES

Functions and Stored Procedures

- Used as a means of reusing code
- Consists of Name, Input variables, return/output variables and a function body
- When combined with variables allows the execution of predefined SQL commands with different variables
- Can be combined with client side coding to enable quick and efficient access of a database based on user inputs

Functions

- Syntax of a function

```
CREATE FUNCTION <function name> (<parameters>)
```

```
RETURNS <return data type>
```

```
AS
```

```
BEGIN
```

```
<function body>
```

```
END
```

- Parameter mode of parameters for functions is IN which parameters allow the calling code to pass values into the function

Functions [contd]

- Create a function that returns the number of employees in a given department.

```
CREATE FUNCTION EmpCount (@did varchar(20))
RETURNS int
AS
BEGIN
    Declare @ecount int
    Select @ecount = count(*)
    From works w
    Where w.did = @did
    return @ecount
END
```

Functions [contd]

- Calling the function created previously

```
Declare @result int
```

```
Exec @result= EmpCount 'Admin'
```

```
Print @result
```

Activity 02

1. Create a function to return the total percentage of time a person works given the employee id.

Procedures

- Syntax of a procedure

```
CREATE PROCEDURE <procedure name> (<parameters>)
```

```
AS
```

```
BEGIN
```

```
<procedure body>
```

```
END
```

- Each parameter to a procedure should have a data type and a parameter mode (IN or OUT)
IN: This is the default mode. IN parameters allow the calling code to pass values into the procedure
OUT: OUT parameters allow the procedure to pass values back to the calling code

Procedures [contd]

- Create a procedure to give a salary increment to all the employees by a given percentage from their current salary.

```
CREATE PROCEDURE IncreaseSalary (@percentage float)
AS
BEGIN
    Update Employee
    Set salary = salary + salary*(@percentage/100)
END
```

- Call the procedure.

```
Exec increaseSalary 10
```

Procedures [contd]

- Create a procedure that outputs the minimum and maximum salary for a given department.

```
CREATE PROCEDURE getDetails (@did varchar(20), @max real output, @min real output)
AS
BEGIN
    Select @max = MAX(e.salary), @min = MIN(e.salary)
    From Employee e, Works w, Dept d
    Where d.did=w.did and w.eid=e.eid and d.did=@did
END
```

Procedures [contd]

- Call the procedure

```
Declare @maxSal real, @minSal real
```

```
Exec getDetails ('Admin', @maxSal output, @minSal output)
```

```
Print @maxSal
```

```
Print @minSal
```

Activity 03

1. Create a procedure that outputs the name of the manager and his salary in a given department.

Syntax Comparision

Stored Procedure

```
CREATE PROCEDURE <procedure  
name> (<parameters> (can be in  
or out))
```

```
AS
```

```
BEGIN
```

```
<procedure body>
```

```
END
```

Function

```
CREATE FUNCTION <function  
name> (<parameters>)
```

```
RETURNS <return data type>
```

```
AS
```

```
BEGIN
```

```
<function body>
```

```
END
```

Comparison

Functions	Procedures
A function has a return type and returns a value.	A procedure does not have a return type. But it returns values using the OUT parameters.
You cannot use a function with Data Manipulation queries. Only Select queries are allowed in functions.	You can use DML queries such as insert, update, select etc... with procedures.
A function does not allow output parameters	A procedure allows both input and output parameters.
You cannot manage transactions inside a function.	You can manage transactions inside a procedure.
You cannot call stored procedures from a function	You can call a function from a stored procedure.
You can call a function using a select statement.	You cannot call a procedure using select statements.

Triggers

- Useful in enforcing business rules and maintaining data integrity
- More powerful than general constraints
- A special type of a procedure that awakes when the data in a specified table is modified
- A trigger is invoked in response to a;
 - DDL statement
 - DML statement

Triggers (cont)

- Syntax

```
CREATE TRIGGER <trigger name>
ON <table | view>
{
    { { FOR | AFTER | INSTEAD OF }
        { [INSERT] [,] [UPDATE] [,] [DELETE] }
    AS
        <SQL Statements>
}
}
```

Triggers [cont]

- **FOR | AFTER**

- DML trigger is fired only when all operations specified in the triggering SQL statement have executed successfully.
- AFTER is the default when FOR is the only keyword specified.
- AFTER triggers cannot be defined on views.

- **INSTEAD OF**

- Trigger is executed instead of the triggering SQL statement, thus overriding the actions of the triggering statements.
- Specifies at most, one INSTEAD OF trigger per INSERT, UPDATE, or DELETE statement can be defined on a table or view.

Triggers [cont]

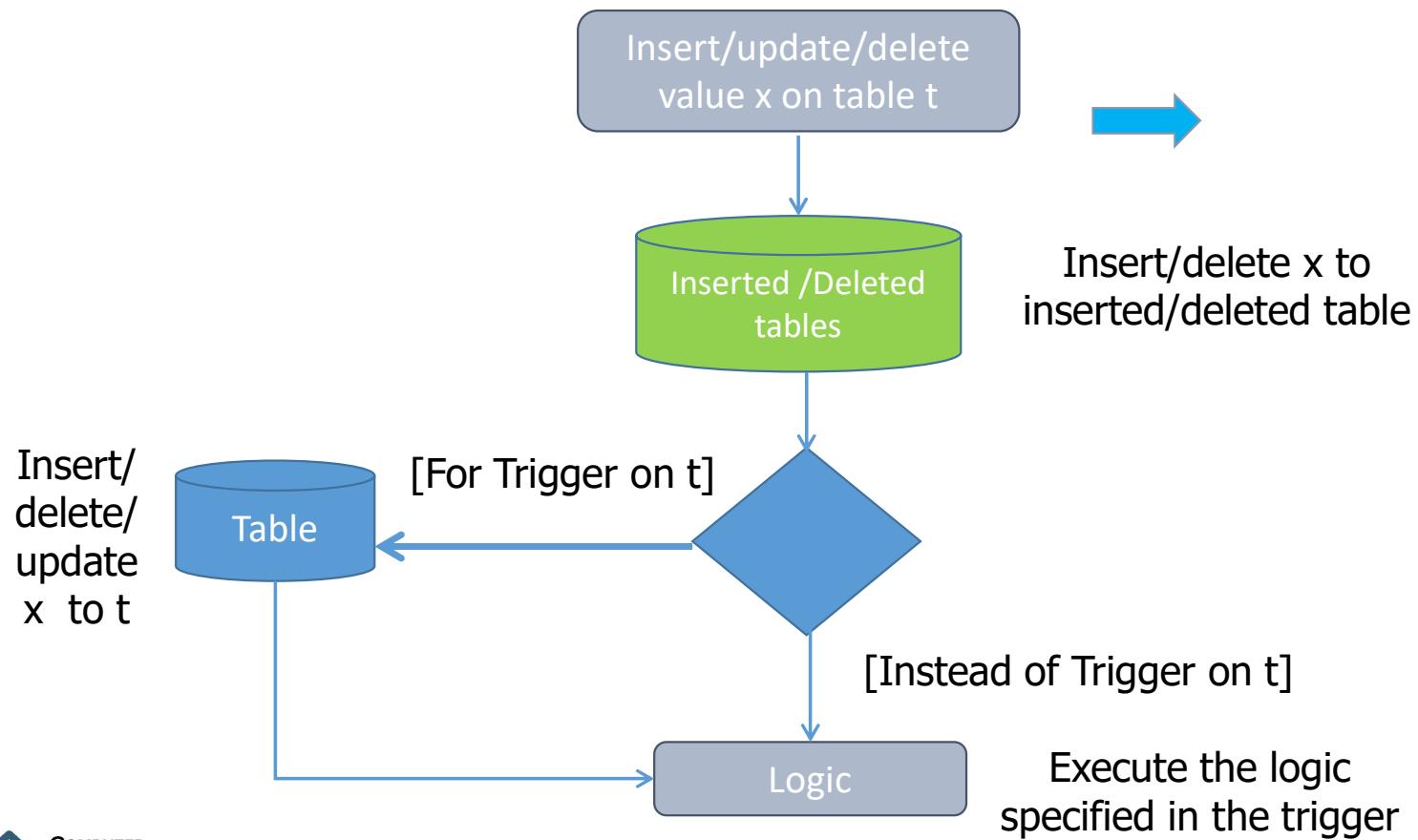
- **[DELETE] [,] [INSERT] [,] [UPDATE]**

- Keywords that specify which data modification statements, when attempted against this table or view, activate the trigger.
- At least one option must be specified.

Triggers [cont]

- When a trigger is executed SQL Server creates two virtual tables called **INSERTED** and **DELETED**.
- The **DELETED** table stores copies of the affected rows during DELETE and UPDATE statements
- The **INSERTED** table stores copies of the affected rows during INSERT and UPDATE statements
 - For example when inserting a record to a table, SQL Server creates a virtual table call **INSERTED** and loads data into the inserted table then executes the trigger statements and writes the related data pages.
- The format of the inserted and deleted tables is the same as the format of the table on which the trigger is defined
- Each column in the inserted and deleted tables maps directly to a column in the base table

Triggers- How it works?



Activity 04

1. Create a trigger to ensure that an employee doesn't work in more than two departments
2. Create a trigger to ensure that no employee has a salary greater than his or her manager.



SLIIT

Discover Your Future

FILE ORGANIZATION & INDEXES



**COMPUTER
SYSTEMS
ENGINEERING**

IE2042- Database Management Systems for Security- Lecture6

Learning Outcomes (LO3)

- ▶ Understand the importance of file organization
- ▶ Identify the types of organizing files
- ▶ Understand the uses of indexes
- ▶ Identify ways to create indexes
- ▶ Understand access methods



FILE ORGANIZATION

Files of Records

- Page or block is OK when doing I/O, but higher levels of DBMS operate on *records*, and *files of records*.
- **FILE:** A collection of pages, each containing a collection of records. Must support:
 - insert/delete/modify record
 - read a particular record (specified using *record id*)
 - scan all records (possibly with some conditions on the records to be retrieved)

File Organization

There are three types

1. Heap File Organization
2. Sequential File Organization
3. Hashing File Organization

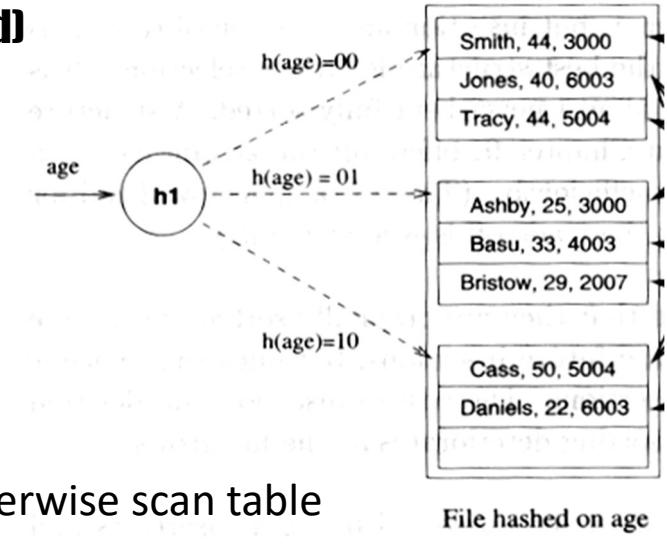
Alternative File Organizations

Many alternatives exist, each ideal for some situation, and not so good in others

- **Heap Files-** Suitable when typical access is a file scan retrieving all records.
 - Search (Equality/Range)- needs to scan the file
 - Insert- at the end of file
 - Delete- search for record and delete record
- **Sequential Files-** Best if records must be retrieved in some order, or only a 'range' of records is needed.
 - Search (Equality/Range)- efficient
 - Insert- finding the position, inserting and move records
 - Delete- search for record, delete and move records

Alternative File Organizations (contd)

- **Hashed Files**- Good for equality selections
 - File is a collection of **buckets**.
 - Bucket = primary page plus zero or more overflow pages.
 - Hashing function **h**: $h(r) = \text{bucket in which record } r \text{ belongs.}$
 - **h** looks at only some of the fields of r , called the search fields.
 - Search (Equality)- good for equality (if based on search key). Otherwise scan table
 - Search (Range)- needs to scan the file
 - Insert- search for primary bucket (hash) and insert
 - Delete- search for primary bucket (hash) if available, else scan file and delete record



INDEXES

Indexes

- An index on a file **speeds up selections** on the search key fields for the index.
 - Any subset of the fields of a relation can be the search key for an index on the relation
 - Search key is not the same as key (minimal set of fields that uniquely identify a record in a relation)
- Index provides fast access
- Index takes space
 - Need to be careful in creating only useful indexes
- May slow down certain inserts/updates/ deletes (maintain indexes)

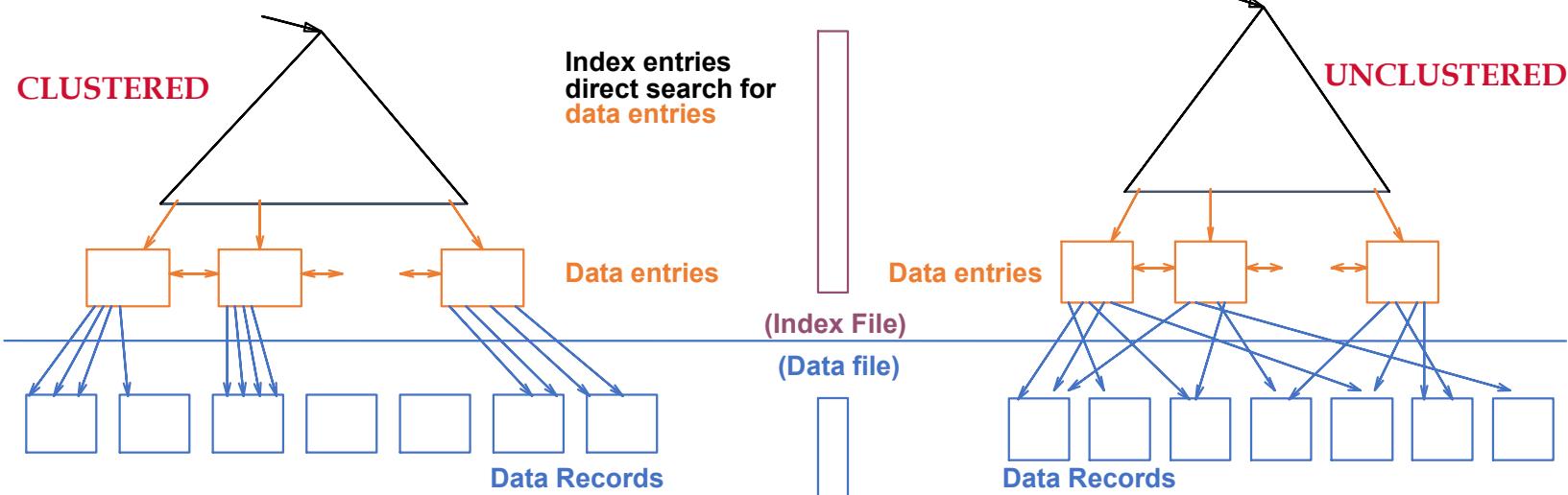
Alternatives for Data Entry k^* in Index

- An index contains a collection of data entries, and supports efficient retrieval of all data entries k^* with a given key value k .
- File of records containing index entries = **Index File**
- Three alternatives: (Methods to connect index data to file records)
 1. Data record with key value k (Alt. 1)
 2. $\langle k, \text{RID of data record with search key value } k \rangle$ (Alt. 2)
 3. $\langle k, \text{list of RIDs of data records with search key } k \rangle$ (Alt. 3)

There are several organization techniques for building index files = **Access Methods**

Properties of Indexes

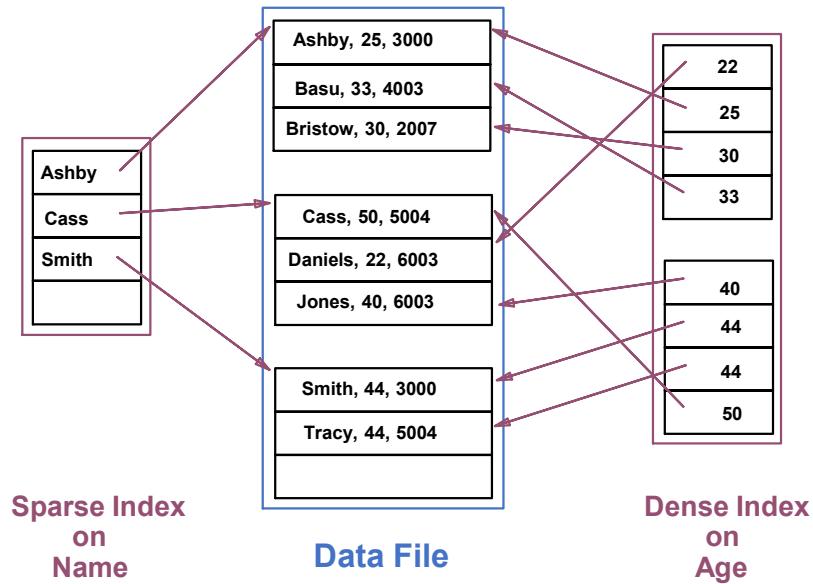
- Clustered vs. Un-Clustered Index



- Can have at most one clustered index per table
- Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!

Properties of Indexes [contd]

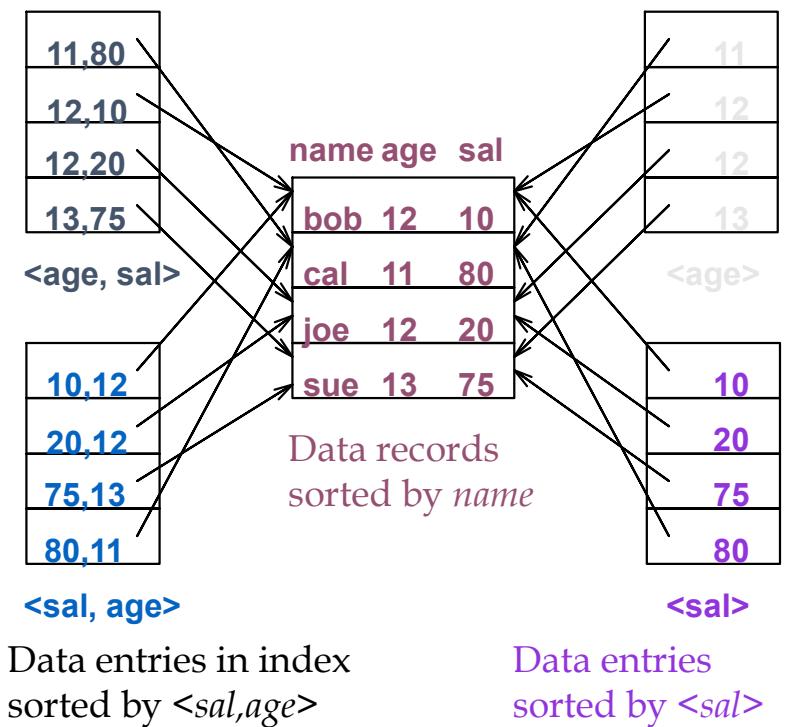
- Sparse vs. Dense Index
- If there is at least one data entry per search key value (in some data record), then dense.
- Alt. 1 always leads to dense index.
- Every sparse index is clustered!
- Sparse indexes are smaller; however, some useful optimizations are based on dense indexes.



Properties of Indexes [contd]

- Primary vs. Secondary Index
- **Primary Index-** Search key contains primary key
- **Unique index-** Search key contains a candidate key
- **Composite Index-** Search on a combination of fields.
 - Equality query: Every field value is equal to a constant value
 - age=20 and sal =75
 - Range query: Some field value is not a constant
 - age =20; or age=20 and sal > 10
- Data entries in index sorted by search key to support range queries

Examples of composite key indexes using lexicographic order.

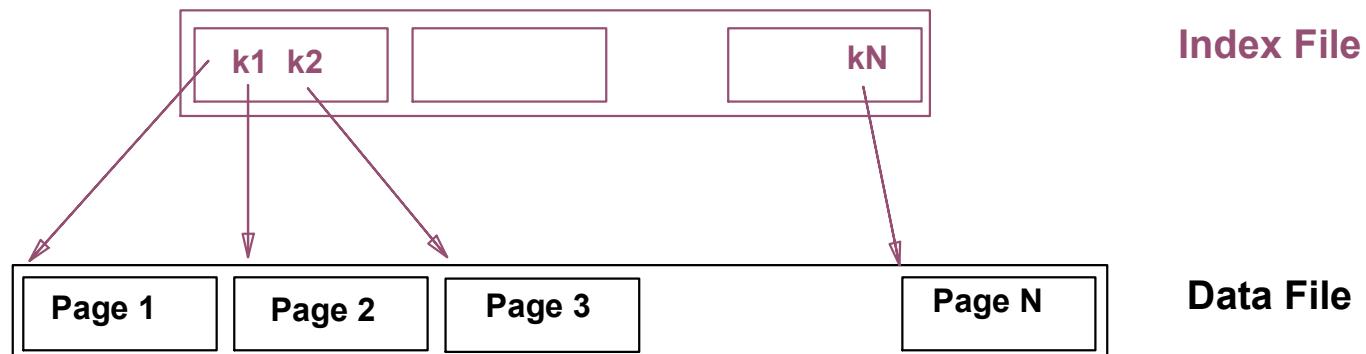


Indexes in SQL

- Index is not a part of SQL-92
- However, all major DBMSs provide facilities for index creation
 - CREATE INDEX...
 - DROP INDEX....
- SQL Server support indexes (clustered and non-clustered indexes)

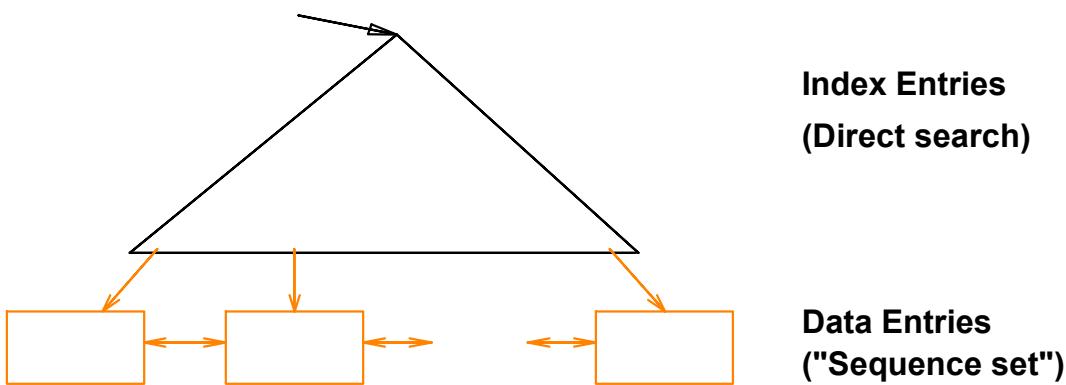
Range Searches

- ``Find all students with gpa > 3.0''
 - If data is in sorted file, do binary search to find first such student, then scan to find others.
 - Cost of binary search can be quite high.
- Simple idea: Create an ‘index’ file.



B+ Tree: The Most Widely Used Index

- Insert/delete at $\log_F N$ cost; keep tree *height-balanced*. (F = fanout, N = # leaf pages)
- Minimum 50% occupancy (except for root). Each node (except root) contains $d \leq m \leq 2d$ entries. The parameter d is called the *order* of the tree.
- Supports equality and range-searches efficiently.

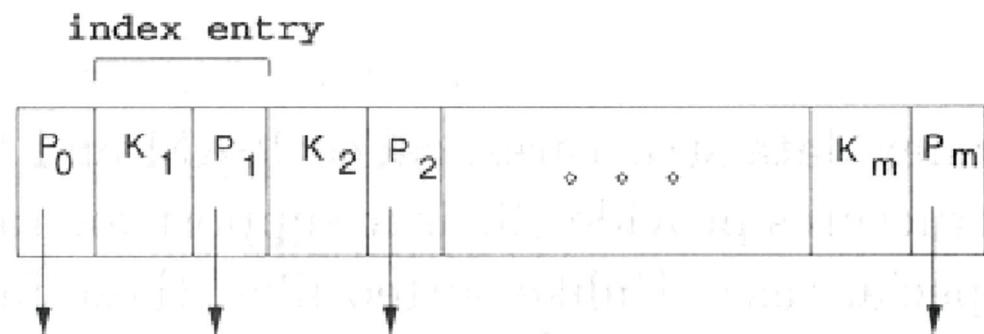


B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
 - average fanout = 133
- Typical capacities:
 - Height 4: $133^4 = 312,900,700$ records
 - Height 3: $133^3 = 2,352,637$ records
- Can often hold top levels in buffer pool:
 - Level 1 = 1 page = 8 Kbytes
 - Level 2 = 133 pages = 1 Mbyte
 - Level 3 = 17,689 pages = 133 MBytes

B+ Tree...

- Search begins at root, and key comparisons direct it to a leaf
- Each Node has search keys (K_i) and pointers (P_i).
- P_i points to a sub-tree in which all key values K are such that $K_i \leq K < K_{i+1}$

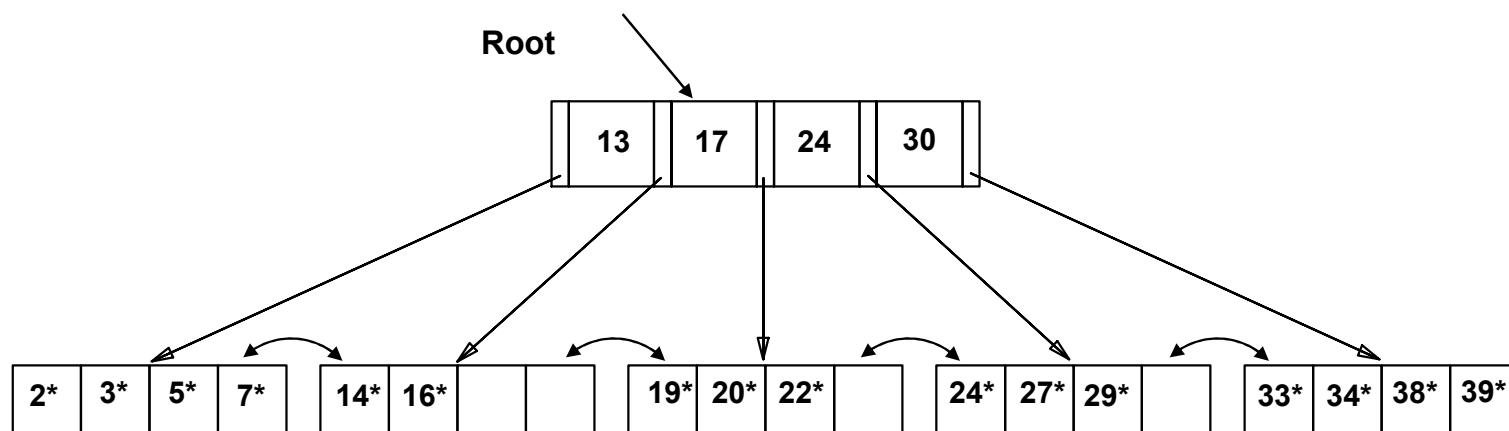


Search

```
func tree_search (nodepointer, search key value K) returns nodepointer
// Searches tree for entry
if *nodepointer is a leaf, return nodepointer;
else,
    if K < K1 then return tree_search(Po, K);
else,
    if K ≥ Km then return tree_search(Pm, K) // m = # entries
    else,
        find i such that Ki ≤ K < Ki+1;
        return tree_search(Pi, K)
    end if
end if
```

Example B+ Tree...

- Search for 5^* , 15^* , all data entries $\geq 24^*$...



Inserting a Data Entry into a B+ Tree

Find correct leaf L .

Put data entry onto L .

If L has enough space, *done!*

Else, must split L (into L and a new node L_2)

Redistribute entries evenly, copy up middle key.

Insert index entry pointing to L_2 into parent of L .

This can happen recursively

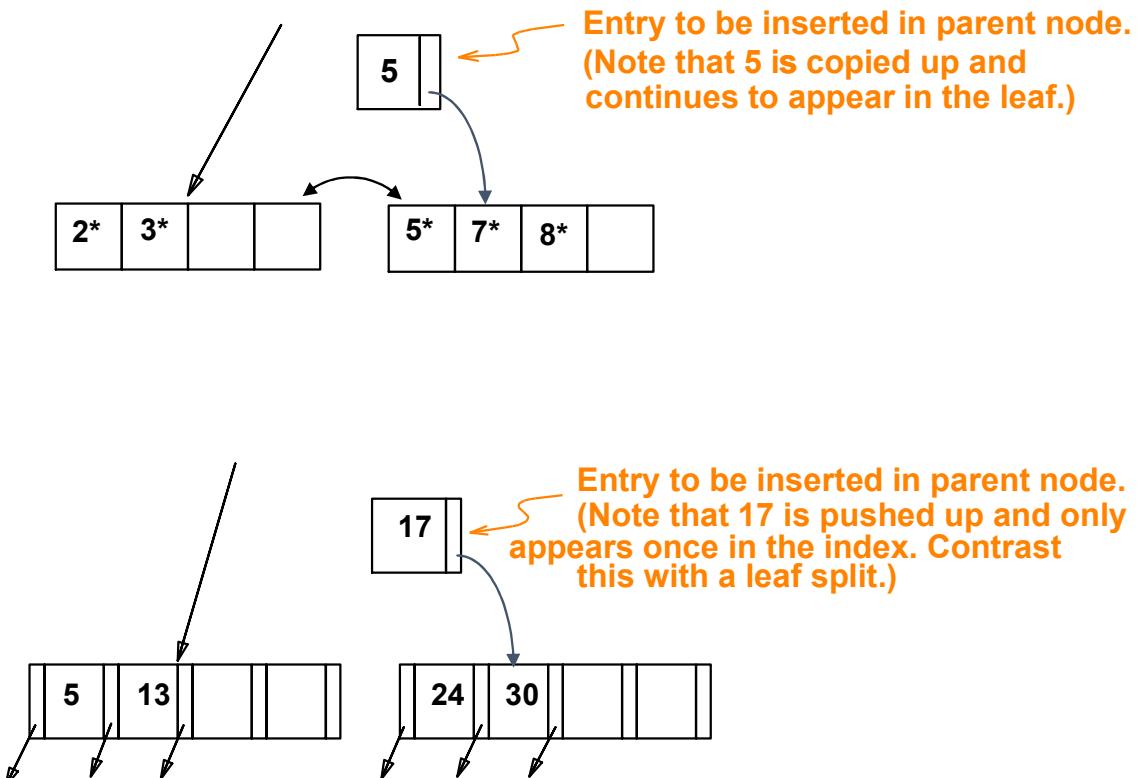
To split index node, redistribute entries evenly, but push up middle key. (Contrast with leaf splits.)

Splits “grow” tree; root split increases height.

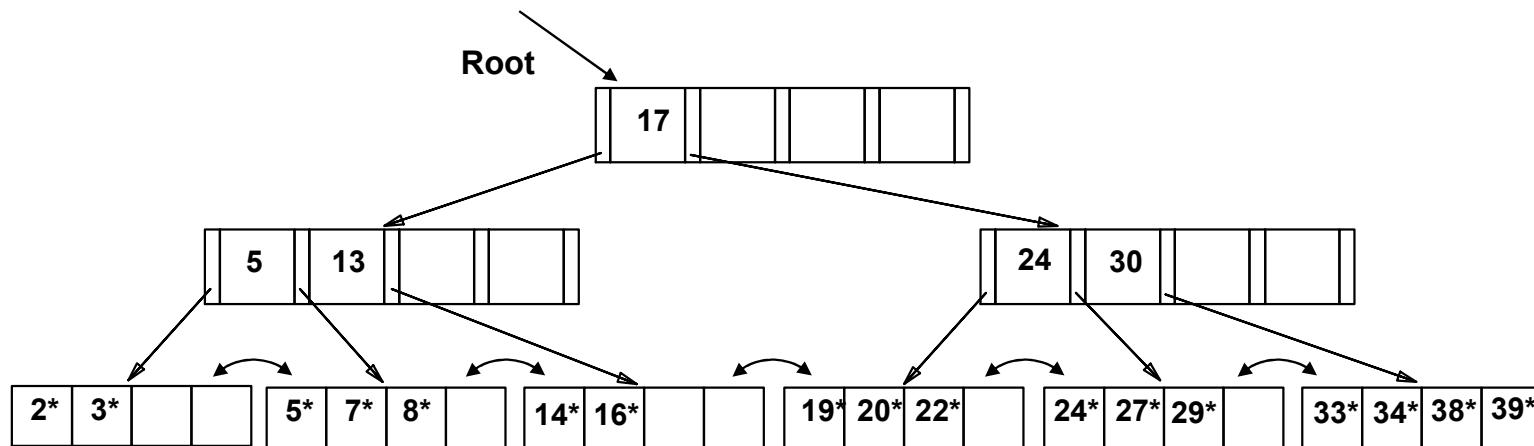
Tree growth: gets wider or one level taller at top.

Inserting 8* into Example B+ Tree

- Observe how minimum occupancy is guaranteed in both leaf and index pg splits.
- Note difference between *copy-up* and *push-up*; be sure you understand the reasons for this.



Example B+ Tree After Inserting 8*

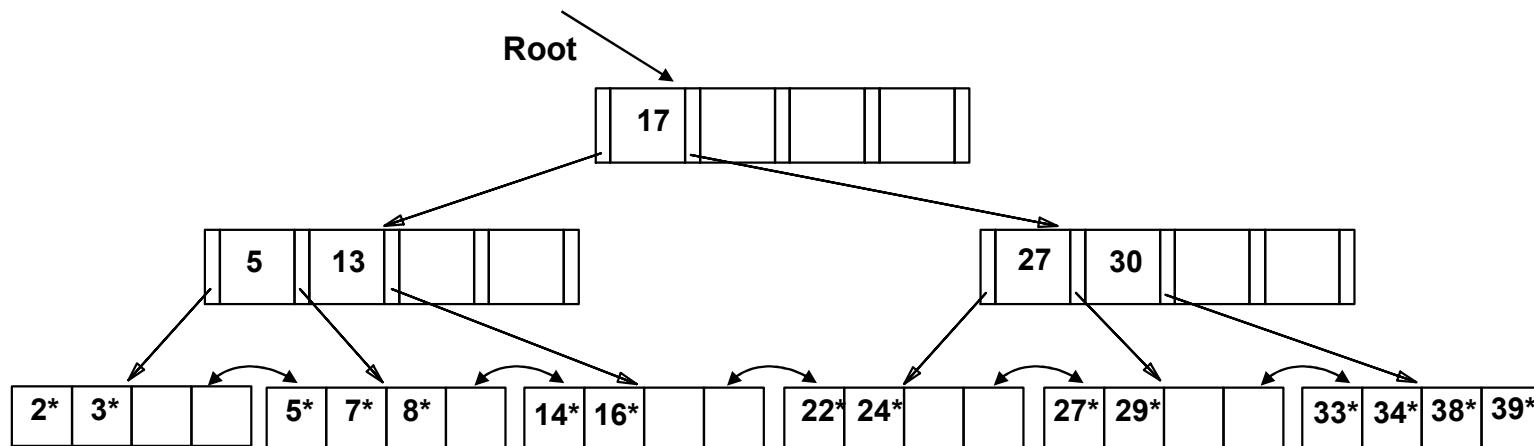


- Notice that root was split, leading to increase in height.
- In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.

Deleting a Data Entry from a B+ Tree

- Start at root, find leaf L where entry belongs.
- Remove the entry.
 - If L is at least half-full, *done!*
 - If L has only **d-1** entries,
 - Try to *re-distribute*, borrowing from *sibling* (*adjacent node with same parent as L*).
 - If re-distribution fails, *merge* L and sibling.
- If merge occurred, must delete entry (pointing to L or sibling) from parent of L .
- Merge could propagate to root, decreasing height.

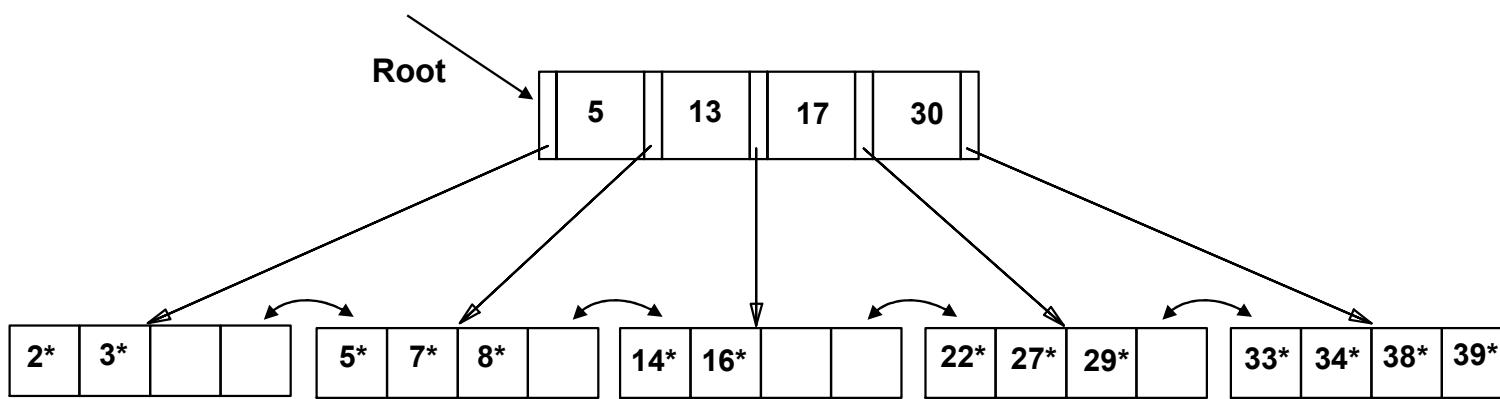
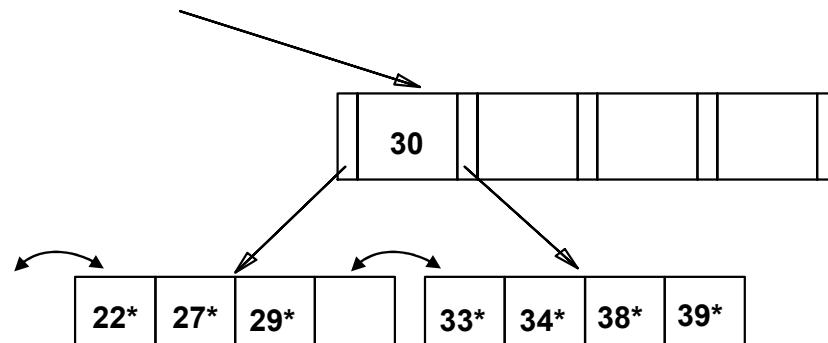
Example Tree After (Inserting 8*, Then) Deleting 19* and 20* ...



- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.

... And Then Deleting 24*

- Must merge.
- Observe '*toss*' of index entry (on right), and '*pull down*' of index entry (below).



Duplicates in B+ Trees...

- We have ignored duplicates so far...
- Alternatives...
 - Overflow leaf pages
 - Duplicate values in the leaf pages
 - Make unique key values (by adding rowid's)
 - Preferred approach by many DBMSs



SLIIT

Discover Your Future

QUERY PROCESSING



**COMPUTER
SYSTEMS
ENGINEERING**

IE2042- Database Management Systems for Security- Lecture 9

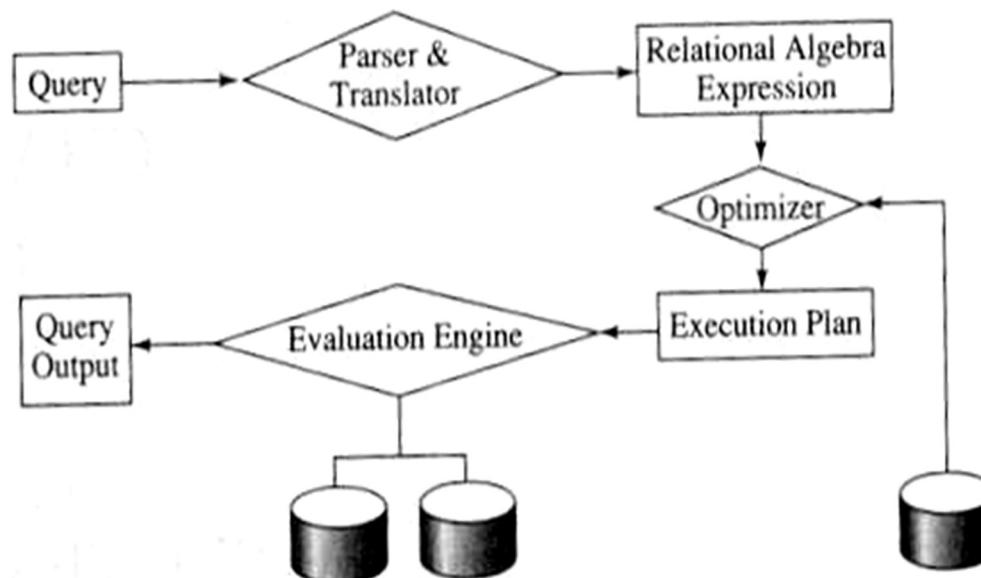
Learning Outcomes (LO3)

- ▶ Understand the steps in query processing
- ▶ Identifying the purpose of using query optimization
- ▶ Understand the relationship between database performance and optimization
- ▶ Learn to design a query plan



Steps in Query Processing

- The steps involved in query processing include...
 1. Parsing and translation
 2. Optimization
 3. Evaluation



Parsing & Translation

- Parsing and translation step **verifies the correctness** of the query and **converts the query into an internal form** (usually an extended relational algebra expression)
- Converts to relational algebraic expressions

Optimization

- Next, an **efficient execution strategy for retrieving the results** of the query from the database files is generated. This step is called query optimization
- This is the **heart of query processing** in a relational database system
- The evaluation engine executes the query according to the chosen plan

Parsing & Translation

- The first step in query processing is to convert the query into a form that can be executed

Consider the following schema

S(sno, sname, status, city)

P(pno, pname, colour, weight)

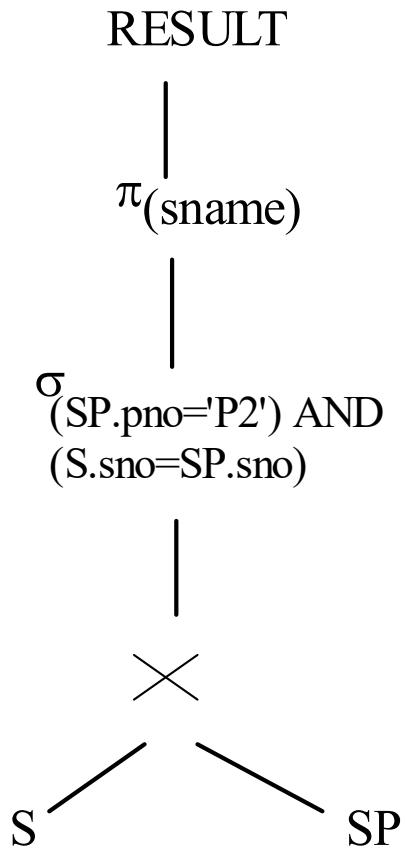
SP(sno, pno, qty)

Parsing & Translation

```
SELECT      s.sname  
FROM        S, SP  
WHERE       S.sno = SP.sno AND SP.pno = 'P2'
```

We can express this query as a relational algebra as follows...

Parsing & Translation



Parsing & Translation

- Usually, a SELECT-FROM-WHERE-GROUP BY called a **query block** is converted to an extended relational algebra expression
- There can be many query blocks (i.e. with nested queries) in a complex query

Why Query Optimization?

- Consider the following number of tuples for S and SP tables

S	-	100 pages
SP	-	10000 pages

- Cost for computing Cartesian Product:

- read 10,000 * 100 pages
 - write 1,000,000 pages (intermediate result)

- Selection

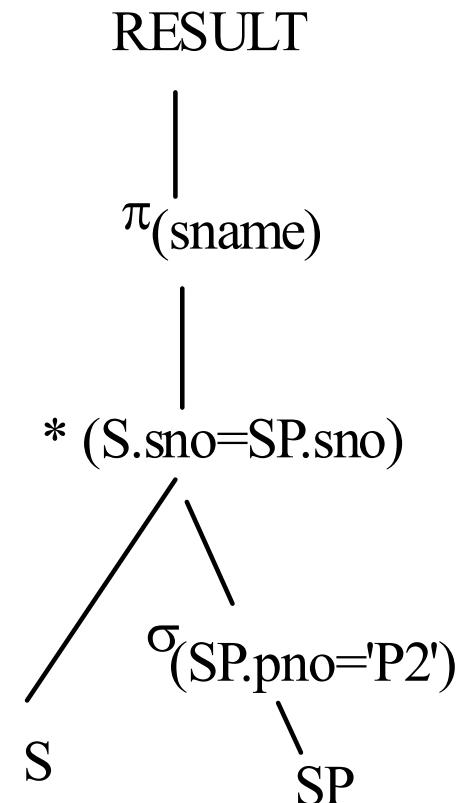
- read 1,000,000 pages
 - keep 50 tuples (assuming 50 tuple SP.pno = 'P2')

- Total number of disk I/O ~ 3,000,000 disk I/Os

Optimization

- Consider the following relational algebra, which produces the same result
- Selection operator
 - Read 10,000 pages
 - Keep the result, 50 tuples in memory
- Join with S
 - Read 100 pages

Total cost 10,100 disk I/Os



Heuristic Optimization?

- A query can have many equivalent query trees
- Optimizer must find an efficient query plan to execute
- Heuristic rules are used for algebraic optimization

Equivalence Rules

- To transform a relational algebra expression to an equivalent efficient query expression, certain equivalence rules are used.

Equivalence Rules

- Conjunctive selection operations can be deconstructed into a sequence of individual selections. This transformation is referred to as a cascade of σ

$$\sigma_{C1 \wedge C2}(E) = \sigma_{C1}(\sigma_{C2}(E))$$

- Selection operations are commutative

$$\sigma_{C1}(\sigma_{C2}(E)) = \sigma_{C2}(\sigma_{C1}(E))$$

- Cascade of Π

$$\Pi_{L1}(\Pi_{L2}(\dots(\Pi_{Ln}(E)\dots)) = \Pi_{L1}(E)$$

Equivalence Rules

- Selections can be combined with Cartesian products and theta joins
 - a.

$$\sigma_{\theta}(E_1 \times E_2) = E_1 \bowtie_{\theta} E_2$$

- This expression is just the definition of the theta join
 - b.

$$\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$$

Etc.

Equivalence Rules

- Equivalence rules states that two expressions are equal
- It does not state, which one is better
- A large number of plans are possible! Estimating the cost for each is expensive
- The optimizer uses heuristic rules to prune the plan space (reduce the number of plans to be considered)
 - E.g. Bring selects down, avoid cartesian products, etc.

Indexes and Cost of Query Plans

- Using an index does not necessarily mean efficient query plan
- Can you think of an instance where using an index is inefficient?
- Query optimizer estimates costs to compare different execution plans

Cost Estimation

- Execution plan has
 - A set of relational algebra operators (query plan) to obtain the result of the query
 - Algorithm used to evaluate each relational algebra operators
- Some relational algebra operators have many possible ways (algorithms)
- Costs may differ significantly based on the chosen algorithm
- We will study some algorithms used for simple selections and joins

Schema for Examples

Sailors (*sid: integer*, *sname: string*, *rating: integer*, *age: real*)

Reserves (*sid: integer*, *bid: integer*, *day: dates*, *rname: string*)

Reserves:

- Each tuple is 40 bytes long, 100 tuples per page, 1000 pages
- Sailors:
 - Each tuple is 50 bytes long, 80 tuples per page, 500 pages

Simple Selections

- Of the form $\sigma_{R.attr \ op \ value}(R)$
- Size of result approximated as *size of R * reduction factor*; we will consider how to estimate reduction factors later
- With no index, unsorted

Must essentially scan the whole relation; cost is M (No. of pages in R)

- With an index on selection attribute

Use index to find qualifying data entries, then retrieve corresponding data records.

(Hash index useful only for equality selections.)

```
SELECT *
FROM Reserves R
WHERE R.day < '01-03-2022'
```

Using an Index for Selection

- Cost depends on number of qualifying tuples, and clustering
 - Cost of finding qualifying data entries (typically small) plus cost of retrieving records (could be large w/o clustering)
- In example, assuming uniform distribution of names, about 10% of tuples qualify (100 pages, 10000 tuples). With a clustered index, cost is little more than 100 I/Os; if unclustered, upto 10000 I/Os!

Equality Selections with One Join Column

```
SELECT *
FROM   Reserves R1, Sailors S1
WHERE  R1.sid=S1.sid
```

- In algebra: $R \times S$. Common! Must be carefully optimized
 $R \times S$ is large; so, $R \times S$ followed by a selection is inefficient
- Assume: M tuples in R, p_R tuples per page, N tuples in S, p_S tuples per page.
 - In our examples, R is Reserves and S is Sailors
- *Cost metric*: # of I/Os. We will ignore output costs

Simple Nested Loops Join (SNLJ)

```
foreach tuple r in R do
    foreach tuple s in S do
        if ri == sj then add <r, s> to result
```

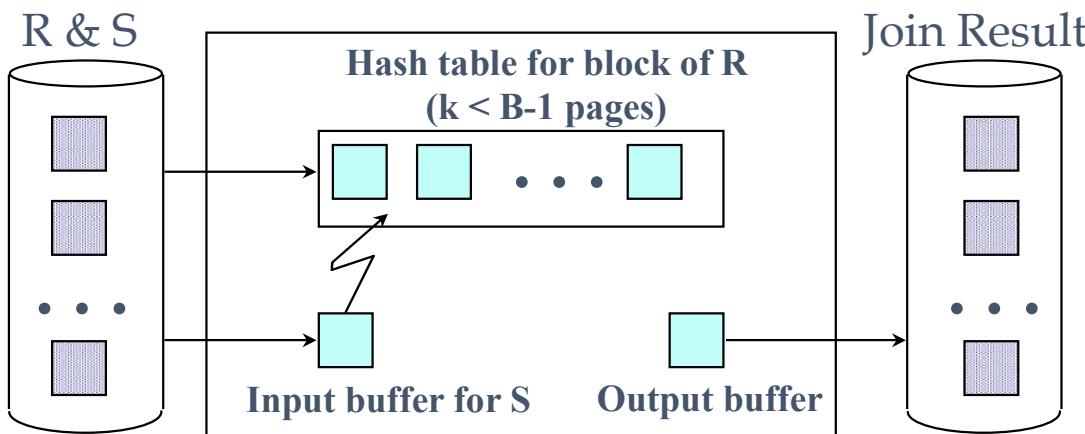
- For each tuple in the *outer* relation R, we scan the entire *inner* relation S
 - Cost: $M + p_R * M * N = 1000 + 100*1000*500$ I/Os

Page-oriented Nested Loop Join (PONLJ)

- Page-oriented Nested Loops join: For each *page* of R, get each *page* of S, and write out matching pairs of tuples $\langle r, s \rangle$, where r is in R-page and S is in S-page
 - Cost: $M + M*N = 1000 + 1000*500$
 - If smaller relation (S) is outer, cost = $500 + 500*1000$

Block Nested Loops Join (BNLJ)

- Use one page as an input buffer for scanning the inner S, one page as the output buffer, and use all remaining pages to hold ``block'' of outer R
 - For each matching tuple r in R-block, s in S-page, add $\langle r, s \rangle$ to result. Then read next R-block, scan S, etc.



Examples of Block Nested Loops

- Cost: Scan of outer + #outer blocks * scan of inner
 - #outer blocks = $\lceil \# \text{ of pages of outer} / \text{blocksize} \rceil$
 - Block size = available buffers - 2
- With Reserves (R) as outer, and 100 pages of R:
 - Cost of scanning R is 1000 I/Os; a total of 10 blocks
 - Per block of R, we scan Sailors (S); 10*500 I/Os
 - If space for just 90 pages of R, we would scan S 12 times
- With 100-page block of Sailors as outer:
 - Cost of scanning S is 500 I/Os; a total of 5 blocks
 - Per block of S, we scan Reserves; 5*1000 I/Os

Index Nested Loops Join (INLJ)

```
foreach tuple r in R do  
    foreach tuple s in S where ri == sj do  
        add <r, s> to result
```

- If there is an index on the join column of one relation (say S), can make it the inner and exploit the index
 - Cost: $M + (M * p_R) * \text{cost of finding matching S tuples}$
- For each R tuple, cost of probing S index is about **1.2 for hash index, 2-4 for B+ tree**. Cost of then finding S tuples (assuming Alt. (2) or (3) for data entries) depends on clustering
 - Clustered index: 1 I/O (typical), unclustered: upto 1 I/O per matching S tuple

Sort-Merge Join ($R \bowtie S$)

i=
j

- Sort R and S on the join column, then scan them to do a “merge” (on join col.), and output result tuples.
 - Advance scan of R until current R-tuple \geq current S tuple, then advance scan of S until current S-tuple \geq current R tuple; do this until current R tuple = current S tuple
 - At this point, all R tuples with same value in R_i (*current R group*) and all S tuples with same value in S_j (*current S group*) *match*; output $\langle r, s \rangle$ for all pairs of such tuples
 - Then resume scanning R and S

Example of Sort-Merge Join

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

<u>sid</u>	<u>bid</u>	<u>day</u>	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

- R is scanned once; each S group is scanned once per matching R tuple. (Multiple scans of an S group are likely to find needed pages in buffer.)
- Cost: $O(M \log M) + O(N \log N) + (M+N)$
 - The cost of scanning, $M+N$, could be $M*N$ (very unlikely!)

Cost-based Optimization

- The fact that there are many algorithms for relational operators, choosing a good query plan using heuristics is not enough
- We can calculate the cost of each candidate query tree with the possible algorithms for each operator (& the difference can be significant)
- To compare such query plans, cost-based optimization techniques are used

Cost Estimation

- For each plan considered, must estimate cost:
 - Must estimate *cost* of each operation in plan tree
 - Depends on input cardinalities
 - We've already discussed how to estimate the cost of operations (sequential scan, index scan, joins, etc.)
 - Must estimate *size of result* for each operation in tree!
 - Use information about the input relations
 - For selections and joins, assume independence of predicates

Statistics and Catalogs

- Need information about the relations and indexes involved. *Catalogs* typically contain at least:
 - # tuples (NTuples) and # pages (NPages) for each relation
 - # distinct key values (NKeys) and NPages for each index
 - Index height, low/high key values (Low/High) for each tree index



SLIIT

Discover Your Future

Transactions Processing

Learning Outcomes (LO4)

- ▶ Understand the concept of a transaction
- ▶ Understand the properties of a transaction
- ▶ Identify different ways of scheduling transactions
- ▶ Understand the problems of unserializable schedules
- ▶ Determine solutions to handle unserializability



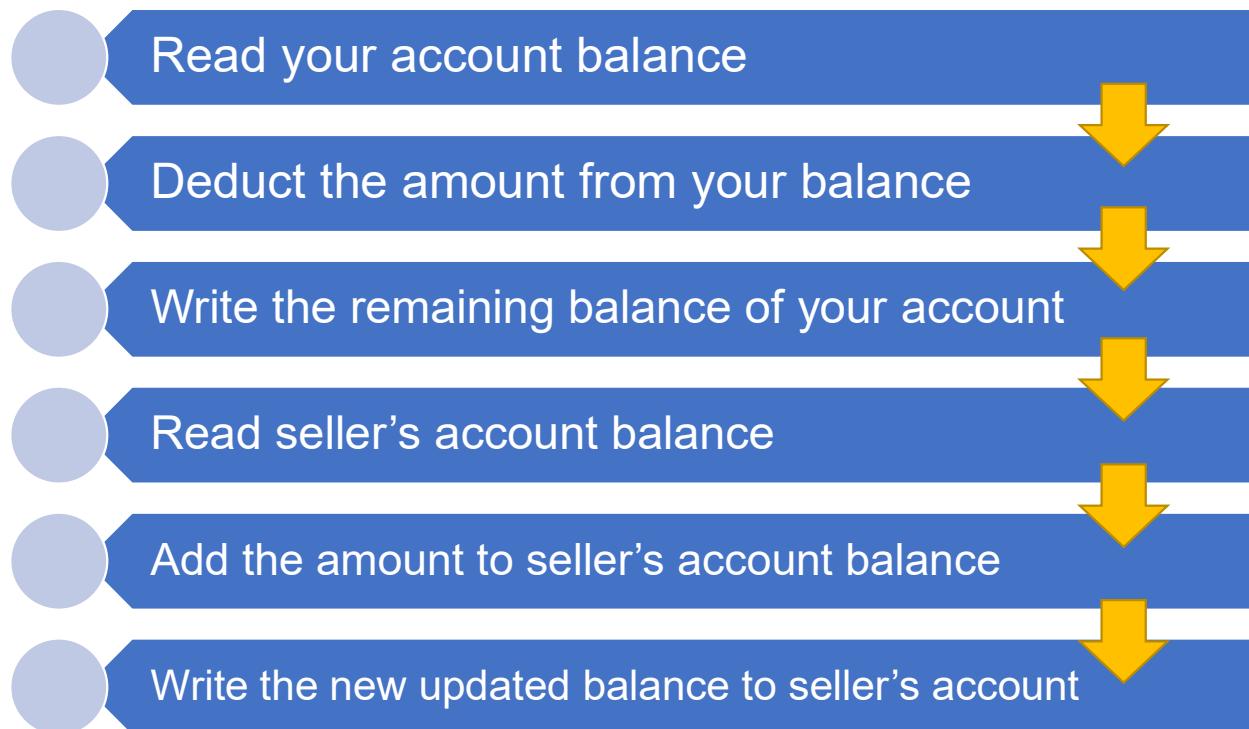
What is a Transaction?

- A user's program may carry out many operations on the data retrieved from the database, but the **DBMS is only concerned about what data is read or written from or to the database**
- An action, or series of actions, carried out by a single user or application program, which reads or updates the contents of the database
- A transaction is a **sequence of reads and writes**.
 - Example: Transferring a balance from Account A to Account B



What is a Transaction?

- You transfer money from your account to a seller's account



Properties of a Transaction

- Atomicity
- Consistency
- Isolation
- Durability

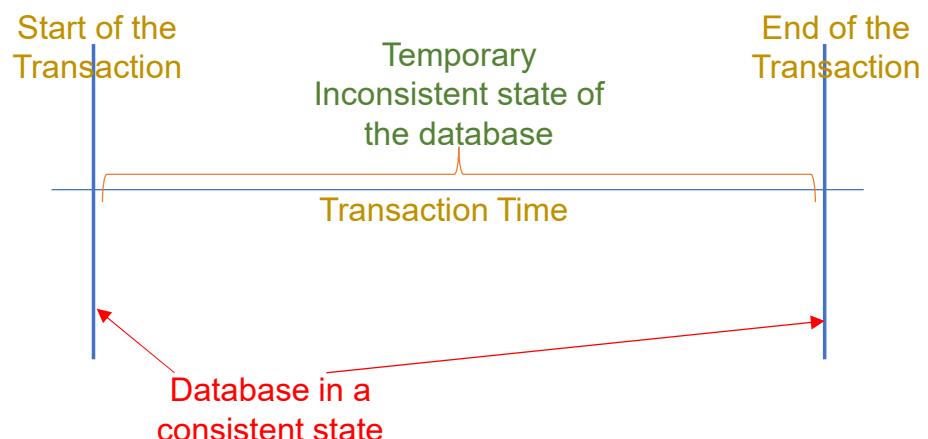


Atomicity

- Atomicity ensures that the transaction is executed as a single unit of work. That is, for every transaction, either all actions within the transaction are carried out or none are (**All or Nothing**)
- A transaction might **commit** after completing all its actions, or it could **abort** (or be aborted by the DBMS) after executing some actions
- In a committed transaction all actions are executed within a DBMS and in an aborted transaction none of the actions are executed
 - DBMS *logs* all actions so that it can *undo* the actions of aborted transactions.
 - Recovery manager is responsible to ensuring atomicity

Consistency

- Consistency ensures that every transaction sees a consistent database instance
- Users/ developers are mainly responsible for ensuring transaction consistency
- Integrity constraints are verified by DBMSs. But program logic must guarantee the consistency of a transactions
 - E.g. Transferring the balance from one account to another. The total amount in both accounts before and after the transaction must be equal



Isolation

- Executed independently of one another
- Isolation property ensures that even though actions of several transactions are interleaved, the net effect is identical to executing all transactions in some serial order
- Concurrency control sub system is responsible to ensure isolation

Durability

- Durability ensures that the transactions survive system crashes and failures
- That is, a committed transaction is always reflected in the database and an uncommitted transaction is always rolled backed after a system crash
- DBMS maintains a log & recovery manager is responsible to ensuring durability

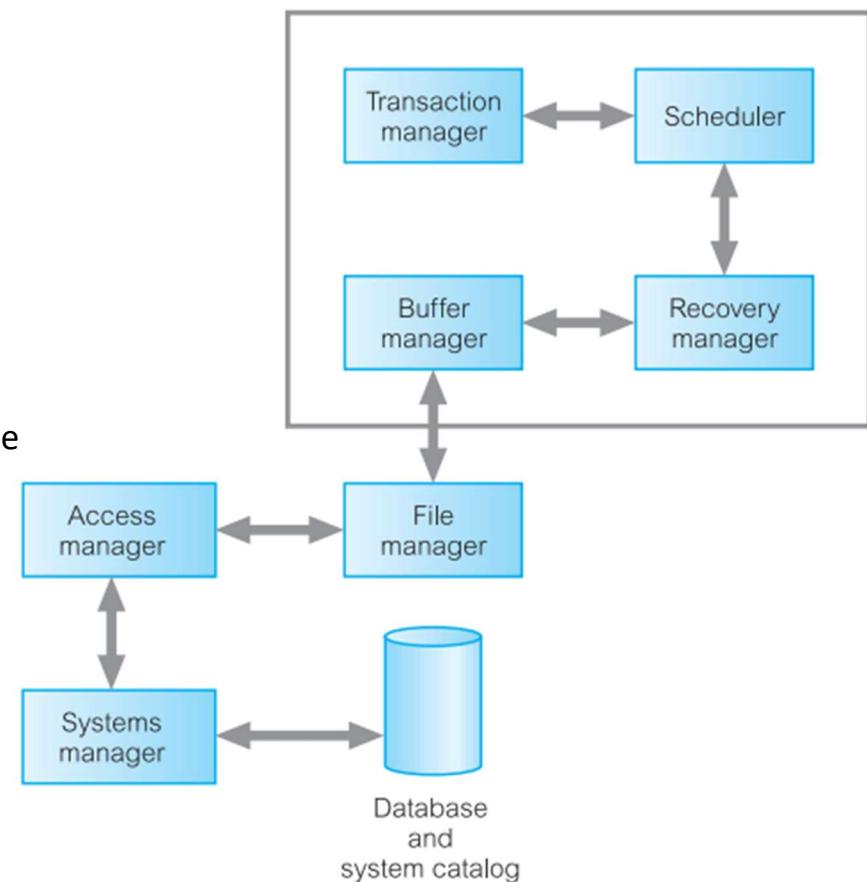
Database Architecture

Transaction manager coordinates transactions on behalf of application programs.

It communicates with the **scheduler**, the module responsible for implementing a particular strategy for concurrency control. The scheduler is sometimes referred to as the **lock manager** if the concurrency control protocol is locking-based. The objective of the scheduler is to maximize concurrency without allowing concurrently executing transactions to interfere with one another, and so compromise the integrity or consistency of the database.

If a failure occurs during the transaction, then the database could be inconsistent. It is the task of the **recovery manager** to ensure that the database is restored to the state it was in before the start of the transaction, and therefore a consistent state.

Finally, the **buffer manager** is responsible for the efficient transfer of data between disk storage and main memory.



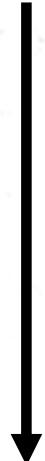
Transactions and Schedules

- A transaction is seen by a DBMS as a series of ordered **actions**
- **Actions** include **reads** and **writes** of database objects
- Notation:
 - Transaction T reading object O : $R_T(O)$
 - Transaction T writing object O : $W_T(O)$
- In addition, the final action of a transaction is **commit** or **abort**.
- Notation:
 - Committing transaction T: $Commit_T$
 - Aborting transaction T : $Abort_T$

Note: We omit the subscript T where unambiguous

Transactions and Schedules

- **Schedule:** A list of actions (*reads, writes, commits, aborts*) from a set of transactions and the order of actions in the schedule is the same as the order of actions in transactions
- Intuitively, a schedule represents an actual or potential execution sequence.



$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(B)$
	$W(B)$
	$R(C)$
	$W(C)$

Concurrent execution of Transactions

- DBMS interleaves actions of different transactions to improve performance
- Motivation for concurrent transactions
 - CPU can process one transaction while another is waiting for a page to be read from disk
 - Interleaving short transactions with longer transactions allows to complete quicker
- Example
 - T1: A transaction calculates the interest for all accounts in the bank (Takes 2 hours)
 - T2: A customer wants to withdraw funds from an account (Takes 1 minute)

Scheduling Transactions

- **Serial schedule:** Schedule that does not interleave the actions of different transactions.
- **Equivalent schedules:** For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
- **Serializable schedule:** A schedule that is equivalent to some serial execution of the transactions.

(Note: If each transaction preserves consistency, every serializable schedule preserves consistency)

Serializability

- Multiple transactions executed concurrently must be equivalent to some serial execution of the transactions in order to preserve consistency
- Order of transactions within the schedule doesn't matter

Anomalies with Interleaved Execution

1. Reading Uncommitted Data (WR Conflicts, “dirty reads”)

Time		T1	T2
		$R(A)$	Initial value
		$W(A)$	Uncommitted
		$R(A)$	Uncommitted
		$W(A)$	Uncommitted
		$R(B)$	Initial value
		$W(B)$	Uncommitted
		Commit	Initial value
		$R(B)$	Uncommitted
		$W(B)$	Uncommitted
		Commit	Initial value

T1 transfers fund

T2 calculates interest

The diagram illustrates interleaved execution between two transactions, T1 and T2. Transaction T1 starts by reading the value of A (R(A)), then writes to A (W(A)). While T1's write is in progress, T2 begins by reading A (R(A)). After T1's write completes, T2 continues to read A (R(A)) and then writes to B (W(B)). Finally, T2 commits its changes to B. When T1 commits, it sees the initial value of B, which is the value that T2 had read before its own write. This results in T1 reading uncommitted data from T2, specifically the value of B before T2's write was completed.

Anomalies with Interleaved Execution

2. Unrepeatable Reads (RW Conflicts)

T1:	R(A),	R(A), W(A), C
T2:	R(A), W(A), C	

Example T1 is incrementing A by 1 and T2 is decrementing A by 1

Anomalies with Interleaved Execution

3. Overwriting Uncommitted Data (WW Conflicts)

T1:	W(A),	W(B), C
T2:	W(A), W(B), C	

Values of A and B are equal.

T1 makes A & B to be 1000.

T2 makes A & B to be 2000.

Unserializability

- The root cause for unserializable schedules (or violation of Isolation property) are conflicts.
 - WR conflict
 - RW conflict
 - WW conflict
- Idea: Avoid conflicts!!!

Lock Based Concurrency Control

- Strict Two-phase Locking (**Strict 2PL**) Protocol:
 - Each transaction must obtain a **S (shared) lock** on object before reading, and an **X (exclusive) lock** on object before writing.
 - All locks held by a transaction are released when the transaction completes
 - If a transaction holds an X lock on an object, no other transaction can get a lock (S or X) on that object.

Strict 2PL

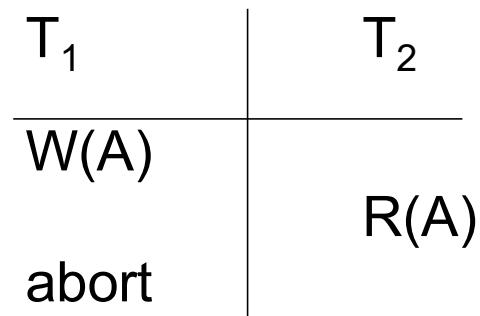
- Strict 2PL does not allow
 - WR conflict
 - RW conflict
 - WW conflict
- How?
- Therefore, Strict 2PL allows only serializable schedules.

Aborting a Transaction

- If a transaction T1 is aborted, all its actions have to be undone
- Not only that, if T2 reads an object last written by T1, then T2 must be aborted as well!
- Some considerations with aborts:
 - Cascading aborts
 - Unrecoverable Schedules

Cascading Abort

- Abort of one transaction causes the abort of another transaction
- Example



T_1 's abort causes abort of T_2 as well

Cascading Abort

- Root cause for cascading abort:
 - WR conflict +
 - Transaction which wrote the data aborts
- Good News: Strict 2PL does not allow cascading aborts!!!
 - How?
- This is also known as **avoiding cascading aborts**

Unrecoverable Schedule

- An unrecoverable schedule:

T_1	T_2
W(A)	
	R(A)
	W(B)
	W(A)
	Commit
abort	

Unrecoverable Schedule

- Root cause for unrecoverable schedule:
 - WR conflict +
 - Transaction which read the data commits before transaction which wrote the data
- Good News: Strict 2PL does not allow unrecoverable schedules!!!
 - How?

The Log

- In order to undo the actions of an aborted transaction, the DBMS maintains a log in which every write is recorded.
- This mechanism is also used to recover from system crashes: all active transactions at the time of the crash are aborted when the system comes back up
- The following actions are recorded in the log:
 - *T_i writes an object*: the old value and the new value.
 - Log record must go to disk before the changed page!
 - *T_i commits/aborts*: a log record indicating this action.
- Log records are chained together by transaction id, so it's easy to undo a specific transaction.
- All log related activities (and in fact, all CC related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.



SLIIT

Discover Your Future

Concurrency Control

Learning Outcomes (LO3)

- ▶ Understand the overheads of locking protocols
- ▶ Identify deadlocks and how to handle them
- ▶ Understand database phantoms and how to handle them
- ▶ Apply index based locking



Overheads of Lock Based CC Protocols

- Locking based concurrency control protocols comes with a numbers of overheads:
 - Handling deadlocks
 - Handling phantoms

Deadlocks

- Deadlock is a cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
 - Deadlock prevention
 - Deadlock detection

Deadlock Prevention

- Assign priorities based on timestamps.
- Assume T1 wants a lock that T2 holds. Two policies are possible:
 - **Wait-Die**: If T1 has higher priority, T1 waits for T2; otherwise T1 aborts
 - **Wound-Wait**: If T1 has higher priority, T2 aborts; otherwise T1 waits
- If a transaction re-starts, make sure it has its original timestamp.
 - Why?

Deadlock Detection

- Create a **wait-for** graph:
 - Nodes are transactions
 - There is an edge from T1 to T2 , if T1 is waiting for T2 to release a lock
- Periodically check for cycles in the waits-for graph
- If deadlock occurs, pick a victim transaction and abort.

Deadlock Detection

Example:

T1: S(A), R(A),

T2:

T3:

T4:

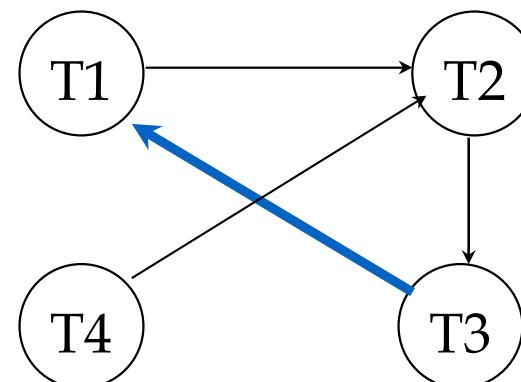
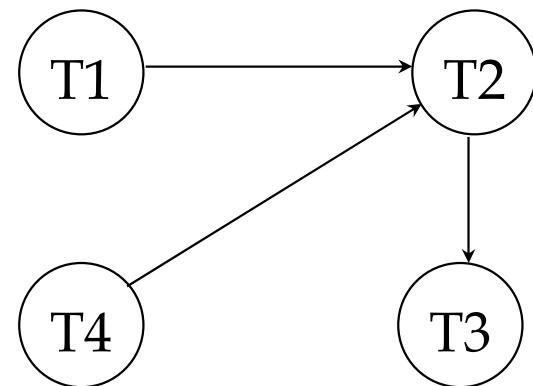
S(B),R(B)
X(B),W(B)

S(C), R(C)

X(C),W(C)

X(A),R(A)

X(B),W(B)



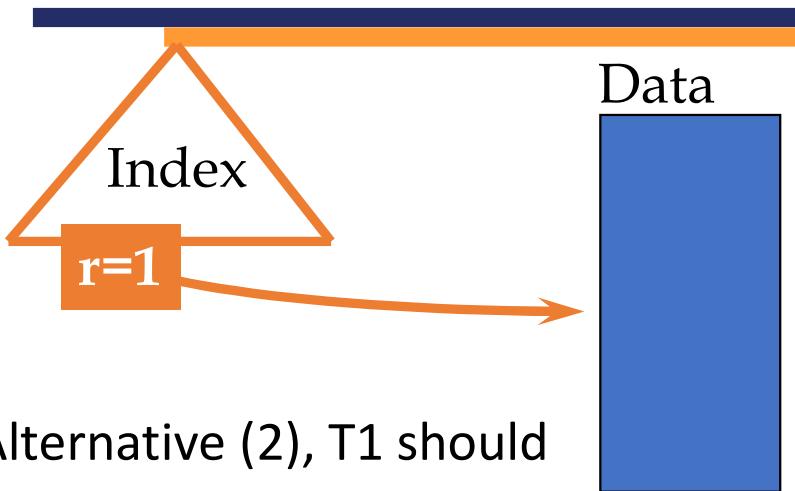
Dynamic Databases and Phantoms

- If we relax the assumption that the DB is a fixed collection of objects, even Strict 2PL will not assure serializability:
 - T1 locks all pages containing sailor records with *rating* = 1, and finds oldest sailor (say, *age* = 71).
 - Next, T2 inserts a new sailor; *rating* = 1, *age* = 96.
 - T2 also deletes oldest sailor with *rating* = 2 (and, say, *age* = 80), and commits.
 - T1 now locks all pages containing sailor records with *rating* = 2, and finds oldest (say, *age* = 63).
- No consistent DB state where T1 is “correct”!

The Problem

- T1 implicitly assumes that it has locked the set of all sailor records with *rating* = 1.
 - Assumption only holds if no sailor records are added while T1 is executing!
 - Need some mechanism to enforce this assumption. (Predicate locking)
- Example shows that conflict serializability guarantees serializability only if the set of objects are fixed!

Index Locking



- If there is a dense index on the *rating* field using Alternative (2), T1 should lock the index page containing the data entries with *rating* = 1.
 - If there are no records with *rating* = 1, T1 must lock the index page where such a data entry *would* be, if it existed!
- If there is no suitable index, T1 must lock all pages, and lock the file/table to prevent new pages from being added, to ensure that no new records with *rating* = 1 are added.

Predicate Locking

- Grant lock on all records that satisfy some logical predicate, e.g. $age > 2 * salary$.
- Index locking is a special case of predicate locking for which an index supports efficient implementation of the predicate lock.
 - What is the predicate in the sailor example?
- In general, predicate locking has a lot of locking overhead.

Locking in B+ Trees

- How can we efficiently lock a particular leaf node?
- One solution: Ignore the tree structure, just lock pages while traversing the tree, following 2PL.
- This has terrible performance!
 - Root node (and many higher level nodes) become bottlenecks because every tree access begins at the root.

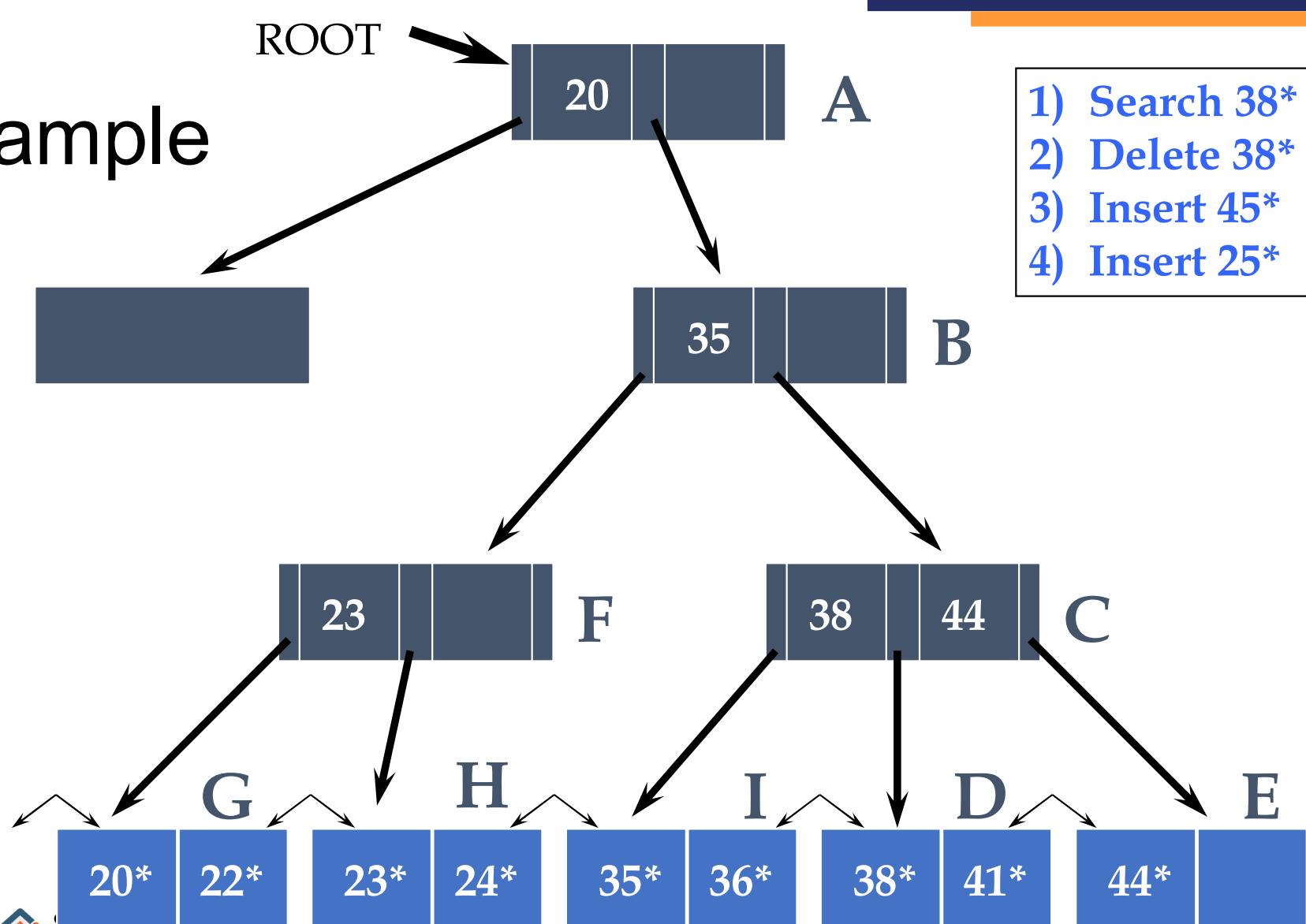
Two Useful Observations

- Higher levels of the tree only direct searches for leaf pages.
- For inserts, a node on a path from root to modified leaf must be locked (in X mode, of course), only if a split can propagate up to it from the modified leaf. (Similar point holds w.r.t. deletes.)
- We can exploit these observations to design efficient locking protocols that guarantee serializability even though they violate 2PL.

A Simple Tree Locking Algorithm

- **Search:** Start at root and go down; repeatedly, S lock child then unlock parent.
- **Insert/Delete:** Start at root and go down, obtaining X locks as needed. Once child is locked, check if it is safe:
 - If child is safe, release all locks on ancestors.
- **Safe Node:** Node such that changes will not propagate up beyond this node.
 - Inserts: Node is not full.
 - Deletes: Node is not half-empty.

Example



Lock Management

- Lock and unlock requests are handled by the **lock manager**
- Lock manager maintains a **lock table**
- **Lock table entry** for an object:
 - Number of transactions currently holding a lock
 - Type of lock held (shared or exclusive)
 - Pointer to queue of lock requests
- Locking and unlocking have to be atomic operations
- Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock

Lock Management

- DBMS also maintains a **transaction table** which contains a pointer to a list of locks held by the transactions
- Transaction requests a lock on an object
 - If shared lock is needed and the request queue is empty and object does not have exclusive lock, then grant & update lock entry
 - If an exclusive lock is needed and the request queue is empty and object is not in shared/exclusive mode, grant & update lock entry
 - Otherwise, add to the request to the request queue



SLIIT

Discover Your Future

DATABASE SECURITY



**COMPUTER
SYSTEMS
ENGINEERING**

IE2042- Database Management Systems for Security- Lecture10

Learning Outcomes (LO4)

- ▶ Understand the three important aspects of security
- ▶ Identify different roles and permissions at server and database level
- ▶ Implement a security policy using access control mechanisms



Database Security

- The data stored in a DBMS is often vital to the business interests of the organization and is regarded as a corporate asset.
- In addition to protecting the intrinsic value of the data, organizations must consider ways to ensure privacy, and to control access to data that must not be revealed to certain groups of users for various reason.

Aspects of Database Security

- There are three main objectives to consider, while designing a secure database application:
 - **Confidentiality:** Information should not be disclosed to unauthorized users.
 - **Integrity:** Only authorized users should be allowed to modify data.
 - **Availability:** Authorized users should not be denied access.



Security Policy and Access Control

- To achieve the objectives, a clear and consistent security policy should be developed
- A security policy specifies who is authorized to do what
- Most users need to access only a small part of the database to carry out their tasks
- Allowing users unrestricted access to all the data can be undesirable
- Two main mechanisms at the DBMS level to control access to data
 - **Discretionary access control**
 - **Mandatory access control**

Discretionary Access Control

- Discretionary access control is based on the concept of access rights, or privileges, and mechanisms for giving users such privileges
- A privilege allows a user to access some data object in a certain manner (e.g., to read or to modify)
- A user who creates a database object such as a table or a view automatically gets all applicable privileges on that object
- The DBMS subsequently keeps track of how these privileges are granted to other users and ensures that at all times only users with the necessary privileges can access an object

Principals, Securables and Permissions

- **Principals :** Individuals, groups, or processes granted access to the SQL Server instance, either at the server level or database level.
 - Server-level principals include logins and server roles.
 - Database-level principals include users and database roles.
- **Securable:** Objects that make up the server and database environment. The objects can be broken into three hierarchical levels:
 - Server-level securables include such objects as databases and availability groups.
 - Database-level securables include such objects as schemas and full-text catalogs.
 - Schema-level securables include such objects as tables, views, functions, and stored procedures.
- **Permissions:** The types of access permitted to principals on specific securables. You can grant or deny permissions to securables at the server, database, or schema level.

Security in SQL Server

- A SQL Server instance contains a hierarchical collection of entities, starting with the server
- Each server contains multiple databases, and each database contains a collection of securable objects
- The SQL Server security framework manages access to securable entities through authentication and authorization
 - **Authentication**
 - verifying the identities of users trying to connect to a SQL Server instance
 - **Authorization**
 - determines which data resources that authorized users can access and what actions they can take.

Security in SQL Server

- Authentication and authorization are achieved in SQL Server through a combination of security principals, securable, and permissions
- Together, these three component types provide a structure for authenticating and authorizing SQL Server users
- You must grant each principal the appropriate permissions it needs on specific securables to enable users to access SQL Server resources

Steps to provide users with access to SQL server resources

1. At the server level, create a login for each user that should be able to log into SQL Server.
 - create Windows authentication logins that are associated with Windows user or group accounts,
 - create SQL Server authentication logins that are specific to that instance of SQL Server.
2. Create user-defined server roles if the fixed server roles do not meet your configuration requirements.
3. Assign logins to the appropriate server roles (either fixed or user-defined).
4. For each applicable server-level securable, grant or deny permissions to the logins and server roles.

Steps to provide users with access to SQL server resources

5. At the database level, create a database user for each login.
 - A database user can be associated with only one server login.
6. Create user-defined database roles if the fixed database roles do not meet your configuration requirements.
7. Assign users to the appropriate database roles (either fixed or user-defined).
8. For each applicable database-level or schema-level securable, grant or deny permissions to the database users and roles.

Scenario

- A software company is assigned to create a software system for a hotel.
- The **project manager** is responsible to develop the system, and he assigns the task of creating the databases (inventory database, sales database, payroll database) to a **senior DBA** (Kamal).
- The senior DBA creates the databases and all logins for the users.
- The senior DBA then assigns two **junior DBAs** (Gayan and Pawan) to look after each database.
- The junior DBA designs the database and assigns the task of creating tables to a **database developer** (Sahan)
- The junior DBA also assigns the task of entering data to some **data entry operators** (Kushani and Chamali)

1. Creating a login using Windows Authentication

- A login is an individual user account for logging into the SQL Server instance.
- Syntax

```
CREATE LOGIN [domain_name\login_name]
FROM WINDOWS
[ WITH DEFAULT_DATABASE = database_name
| DEFAULT_LANGUAGE = language_name ];
```

- Example

```
CREATE LOGIN [win10b\winuser01] FROM WINDOWS
WITH DEFAULT_DATABASE = master, DEFAULT_LANGUAGE = us_english;
```

1. Creating a login using SQL Server

- Syntax

```
CREATE LOGIN [Login Name] -- This is the User that you use for login  
WITH PASSWORD = 'provide_password' MUST_CHANGE,  
CHECK_EXPIRATION = ON, CHECK_POLICY = ON, DEFAULT_DATABASE = [Database Name],  
DEFAULT_LANGUAGE = [Language Name];-- This is Optional
```

- Example

```
CREATE LOGIN sqluser01  
WITH PASSWORD = 'tempPW@56789'  
MUST_CHANGE, CHECK_EXPIRATION = ON,  
DEFAULT_DATABASE = master, DEFAULT_LANGUAGE = us_english;
```

2. Creating and assigning server roles

- With sever roles, logins could be grouped together in order to easily manage server-level permissions.
- SQL Server supports fixed server roles and user-defined server roles.
- With **fixed** server roles logins could be assigned, but permissions cannot be changed.
- For **user-defined** roles logins can be assigned and permission can be changed.

3. Adding logins to fixed server roles

- Assigning logins to sever roles
 - Syntax : ALTER SERVER ROLE server_role_name ADD MEMBER login
 - Server role names :

Role	Description
Sysadmin	Members of the sysadmin fixed server role can perform any activity in the server.
dbcreator	Members of the dbcreator fixed server role can create, alter, drop, and restore any database.
securityadmin	Members of the securityadmin fixed server role manage logins and their properties. They can GRANT, DENY, and REVOKE server-level permissions.

- Ex : ALTER SERVER ROLE diskadmin ADD MEMBER Ted

4. Creating user-defined server roles and permissions

- Creating a user defined sever role and assigning permissions
 - Syntax for creating a server role: CREATE SERVER ROLE role_name
 - Ex: CREATE SERVER ROLE devops;
 - Syntax for granting permissions : GRANT permission TO grantee_principal [WITH GRANT OPTION]
 - Permission Examples :
 - CREATE ANY DATABASE — Create a database on the server.
 - ALTER ANY DATABASE — Create, alter, or drop any login in the instance.
 - ALTER ANY LOGIN — Modify any login.
 - SHUTDOWN — Shut down the server.
 - Ex: GRANT ALTER ANY DATABASE TO devops;

Scenario (contd)

- Senior DBA, Junior DBA, Database Developer and Data entry operator should have logins.
- According to the scenario, Senior dba also should be able to create databases and logins.
- Therefore two methods could be used to create logins
 - All logins could be created by the project manager
 - Project manager could create the login of Senior DBA and Senior DBA could create the other logins. (we will use this approach)

Scenario (contd)

- First create the login for the senior DBA using windows authentication of SQL server authentication as follows:
- Windows authentication

```
CREATE LOGIN PC2\kamal  
FROM WINDOWS WITH DEFAULT_DATABASE = Master
```

- SQL server authentication

```
CREATE LOGIN kamal  
WITH PASSWORD = 'kamal@123', DEFAULT_DATABASE = Master
```

Scenario (contd)

- Now let's give permission to the senior DBA to create databases and logins.
- This could be done using pre-defined sever roles or user defined server roles as follows
- Using pre-defined server roles
`alter server role dbcreator add member amal`
`alter server role securityadmin add member kamal`
- Using user-defined server role
`create server role seniordba`
`grant create any database,alter any login to seniordba`
`alter server role seniordba add member kamal`

5. Database Users

- Logins are created at the server level, while users are created at the database level.
- In other words, a login allows to connect to the SQL Server service and permissions inside the database are granted to the database users, not to the logins.
- The logins will be assigned to server roles (for example, *serveradmin*)
- The database users will be assigned to roles within that database.
- Syntax : CREATE USER user_name FOR LOGIN login_name
 - user_name : The name of the database user that you wish to create.
 - login_name : The Login used to connect to the SQL Server instance

Scenario (contd)

- Senior DBA can now create the databases and create a logins for the junior DBAs who are in-charge of the databases
- Create the login first

```
CREATE LOGIN gayan
```

```
WITH PASSWORD = 'gayan@123', DEFAULT_DATABASE = Inventory
```

- The senior DBA can also create all the other logins for the other roles here
- Now let's create the user for the login so that the user can be given permission at the database level

```
Create user gayan for login gayan
```

6. Creating database roles

- A database role is a group of users that share a common set of database-level permissions.
- As with server roles, SQL Server supports both fixed and user-defined database roles.
- To set up a user-defined database role, you must create the role, grant permissions to the role, and add members to the role (or add members and then grant permissions).

7. Assigning users to fixed database roles

- Some examples of database roles are as follows:

Role	Description
db_owner	Members of the db_owner fixed database role can perform all configuration and maintenance activities on the database, and can also drop the database in SQL Server.
db_securityadmin	Members of the db_securityadmin fixed database role can modify role membership for custom roles only and manage permissions.
db_accessadmin	Members of the db_accessadmin fixed database role can add or remove access to the database
db_ddladmin	Members of the db_ddladmin fixed database role can run any Data Definition Language (DDL) command in a database.

- Syntax for windows authentication :

```
ALTER ROLE db_datawriter ADD MEMBER <domain\username>
```

- Syntax for SQL authentication :

```
ALTER ROLE db_datawriter ADD MEMBER <username>;
```

Scenario (contd)

- The junior DBAs are expected to manage the databases.
- Therefore they could be assigned with the db_owner role by the senior DBA as follows :

```
use inventoryDB
```

```
alter role db_owner add member gayan
```

```
alter role db_securityadmin add member gayan
```

```
alter role db_accessadmin add member gayan
```

Assigning users to user defined database roles

- Creating a user defined role
 - Syntax : CREATE ROLE <role_name>
 - Example : CREATE ROLE student
- Assigning login to the role
 - Syntax : ALTER ROLE <role_name> ADD MEMBER <username>
 - Example : ALTER ROLE student ADD MEMBER Jim

Permissions

- Every SQL Server securable has associated permissions that can be granted to user.
- At the database level they are assigned to database users and database roles.

Permission	Description
ALTER	Grants or denies the ability to alter the existing database.
CREATE TABLE	Grants or denies the ability to create a table.
INSERT	Grants or denies the ability to issue the INSERT command against all applicable objects within the database.
EXECUTE	Grants or denies the ability to issue the EXECUTE command against all applicable objects within the database.
REFERENCES	The REFERENCES permission on a table is needed to create a FOREIGN KEY constraint

Creating and removing permissions

- Creating and removing permissions could be done using GRANT, DENY and REVOKE
 - **GRANT** – Grants permissions on a securable to a principal.
 - **DENY** – Denies a permission to a principal.
 - **REVOKE** – Removes a previously granted or denied permission.

Granting permissions

- Granting permissions to a role/user
 - Ex: **GRANT SELECT TO** GrantSelectRole
 - Ex: **GRANT SELECT, INSERT, UPDATE, DELETE, REFERENCES, EXECUTE ON** inventoryDB **TO** sqluser01;
- Denying permissions to a role/user
 - Ex: **DENY SELECT TO** sqluser02
- Revoking permissions assigned to a user
 - Ex: **REVOKE SELECT TO** GrantSelectRole;

Scenario (contd)

- The database developers need to create tables which would probably have foreign keys.
- Let's create a role named database developer for this
 - create role databaseDev
 - grant create table, alter, references to databaseDev on inventoryDB
 - alter role databaseDev add member Sahan
- Similarly create another role with select and insert for the data Entry operators

Views and Security

- Views can be used to present necessary information (or a summary), while hiding details in underlying relation(s).
- Creator of view has a privilege on the view if (s)he has the privilege on all underlying tables.
- Together with GRANT/REVOKE commands, views are a very powerful access control tool