
CS3402

Database Systems

Review

Assessment

- Coursework -- 40% :
 - ◆ Mid-term -- 25%
 - ◆ Homework assignments (3 times) -- 15%
 - ◆ Lab attendance (At least 5 of the 6 labs) -- 3% *bonus*
(such that coursework is capped to 40%)
- Final examination -- 60%
 - ◆ *Get 30 out of 100 to pass*

Final Exam Info

- **Date:** Dec 13th (**Wednesday**)
- **Time:** 6:30-8:30 pm (2 hours) (Please arrive at least 15 min before the exam starts.)
- **Venue:** Please refer to your assigned venue by ARRO.
- **Coverage:** **Lecture 1-11**
- **Format:** Open-book. The following materials are allowed: *Printed version of lecture/tutorial notes, personal notes, and textbook. (no calculator, no electronic devices).*
- **Mode:** Face-to-face, paper writing
- **Question type:** Similar to practice, homework and midterm

Lecture 1:

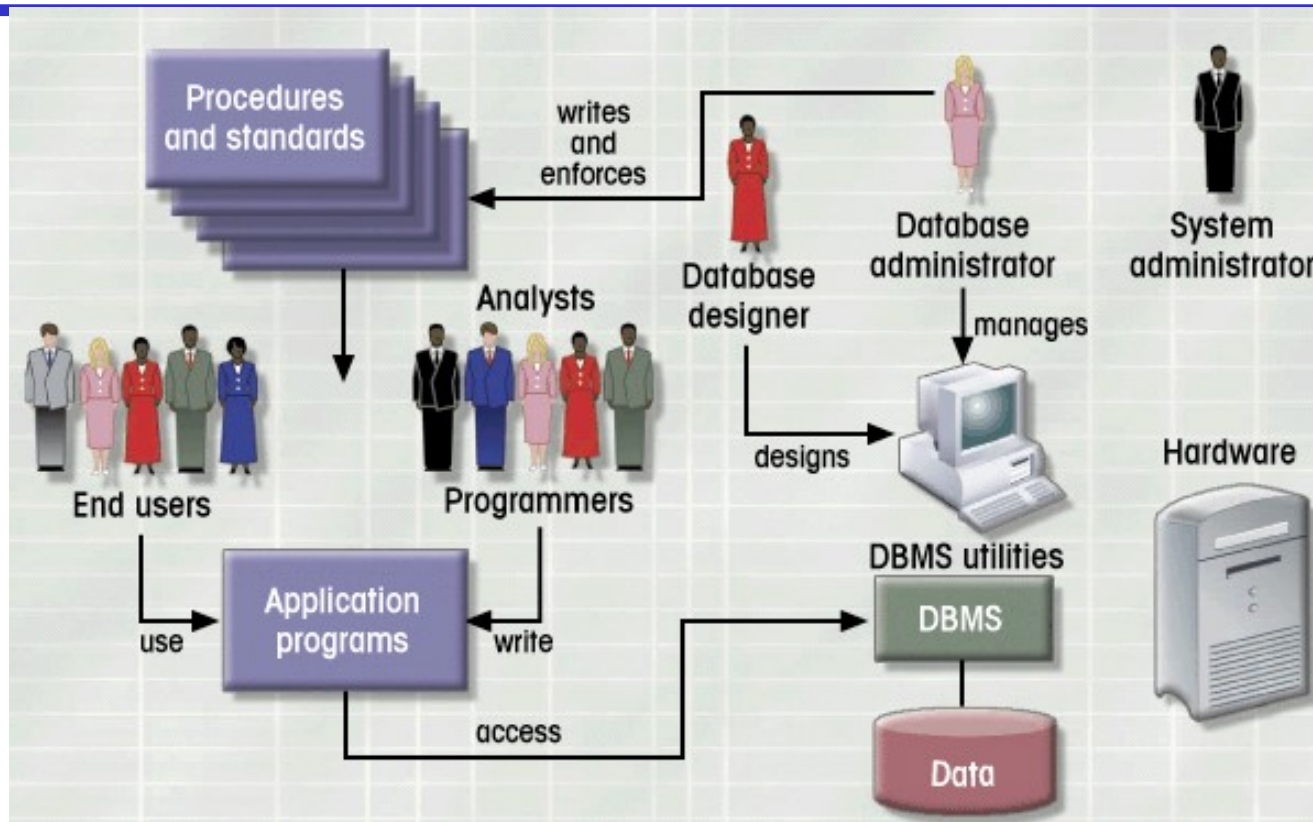
Basic information and ER model

Introduction to DB Systems

- What is a Database (DB)?
 - ◆ A non-redundant, persistent collection of logically-related records/files that are structured to support various processing and retrieval needs.

- Database Management System (DBMS)
 - ◆ A set of software programs for creating, storing, updating, and accessing the data of a DB.
 - ◆ E.g.: Oracle, Mysql, Oceanbase

Database System



- **Hardware**
- **Software**
 - OS
 - DBMS
 - Applications
- **People**
- **Procedures**
- **Data**

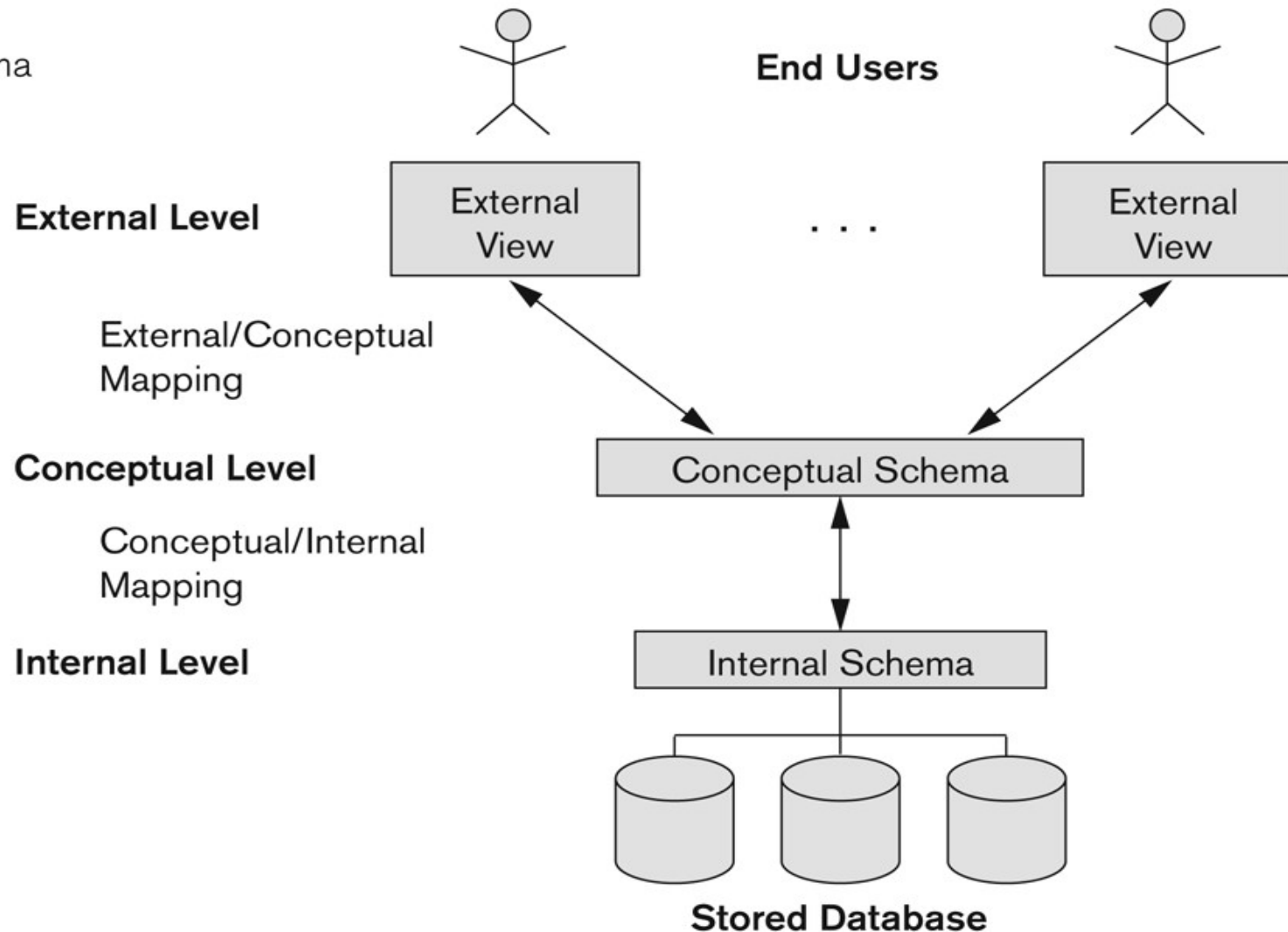
■ Database System

- ◆ an **integrated system** of hardware, software, people, procedures, and data
- ◆ that define and regulate the collection, storage, management, and use of data within a database environment

Data Abstraction: 3-level architecture

Figure 2.2

The three-schema architecture.



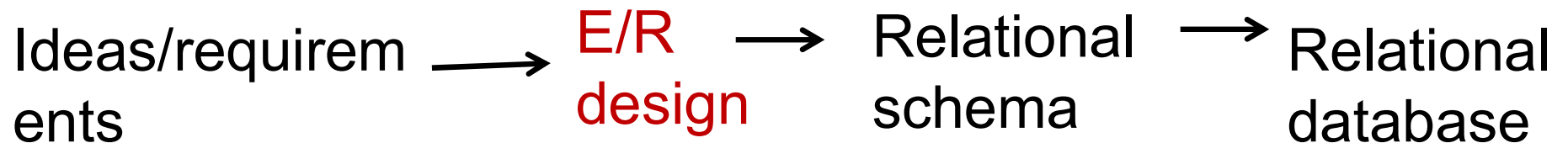
Data Independence

- ◆ the ability to modify a schema definition in one level without affecting a schema in the next higher level
- ◆ there are two kinds (a result of the 3-level architecture):
 - ◆ ***physical data independence***
 - *the ability to modify the physical schema without altering the conceptual schema and thus, without causing the application programs to be rewritten*
 - ◆ ***logical data independence***
 - *the ability to modify the conceptual schema without causing the application programs to be rewritten*

The Entity-Relationship Model

■ Preliminaries

- ◆ Proposed by P. Chen in 1976
- ◆ Direct, easy-to-understand graphical notation
- ◆ Translates readily to relational schema for database design



■ Three basic concepts:

Entity, Attribute, Relationship

ER Model Concepts

■ Entity

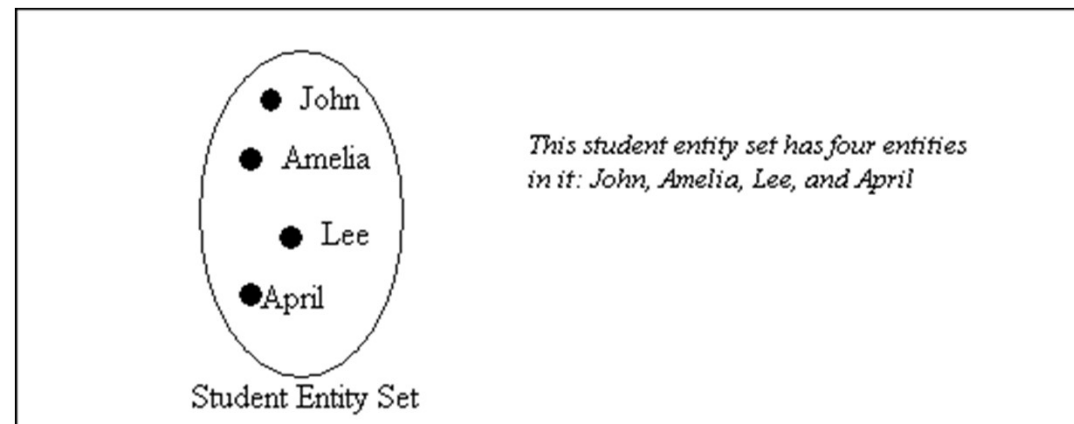
- ◆ a distinguishable object with an independent existence

Example: John Chan, CityU, HSBC, ...

■ Entity Set

- ◆ a set of entities of the same type

Example: Student, University, Bank, ...



ER Model Concepts

- **Attribute**(Property) -- a piece of information describing an entity or a relationship
 - ◆ Example: Name, ID, Address, Sex are attributes of a student entity
 - ◆ Each attribute can take a **value** from a **domain**
Example: Name \in Character String,
ID \in Integer, ...
 - ◆ Formally, an attribute **A** is a function which maps from an entity set **E** into a domain **D**:
$$\mathbf{A}: \mathbf{E} \rightarrow \mathbf{D}$$

Types of Attributes

■ Simple

- ◆ Each entity has a **single atomic value** for the attribute. For example, SSN or Sex, name...

■ Composite

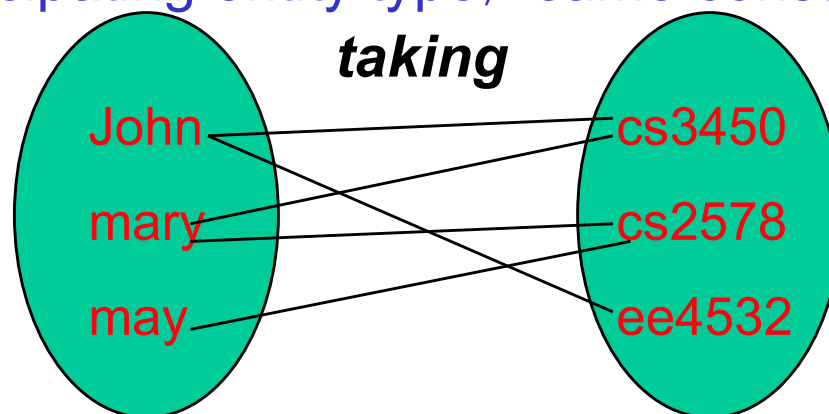
- ◆ The attribute may be composed of several components. For example:
 - ◆ Address(Flat, Block, Street, City, State, Country)
 - ◆ Composition may form a **hierarchy** where some components are themselves composite

■ Multi-valued

- ◆ An entity may have multiple values for that attribute. For example, Color of a CAR or PreviousDegrees of a STUDENT
 - ◆ Denoted as {Color} or {PreviousDegrees}
 - ◆ E.g., {{BSc, 1990}, {MSc, 1993}, {PhD, 1998}}

ER Model Concepts

- **Relationship** -- an association among several entities
 - ◆ Example: Patrick and Eva are friends
Patrick is taking cs3450
- a relationship can carry attributes: properties of the relationship
 - ◆ Example: Patrick takes cs3450 with a grade of B+
- **Relationship Set** -- a set of relationships of the same type (same attribute, same participating entity type, same constraints)
 - ◆ Example:

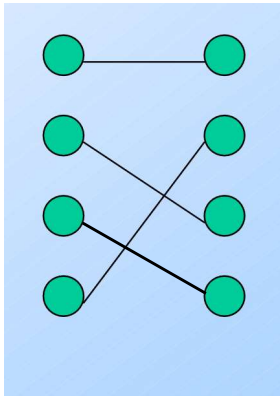


- ◆ Formally, a relationship **R** is a subset of:
 $\{ (e_1, e_2, \dots, e_k) \mid e_1 \in E_1, e_2 \in E_2, \dots, e_k \in E_k \}$

Constraints on relationship

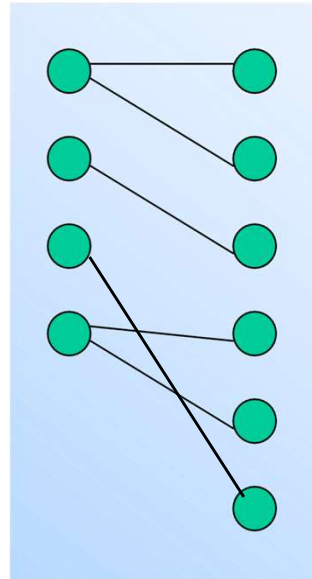
◆ Cardinality

COURSE LECTURER



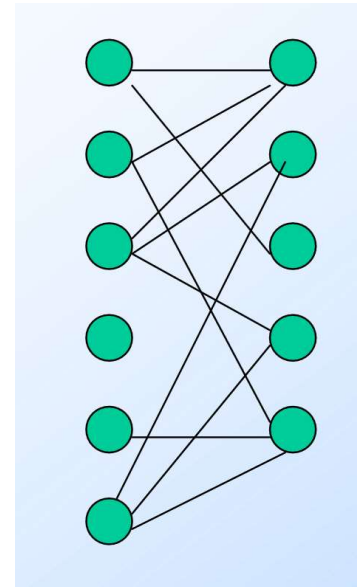
One-to-one(1:1)

COURSE TA



**One-to-many
(1:N)**

STUDENTS COURSE



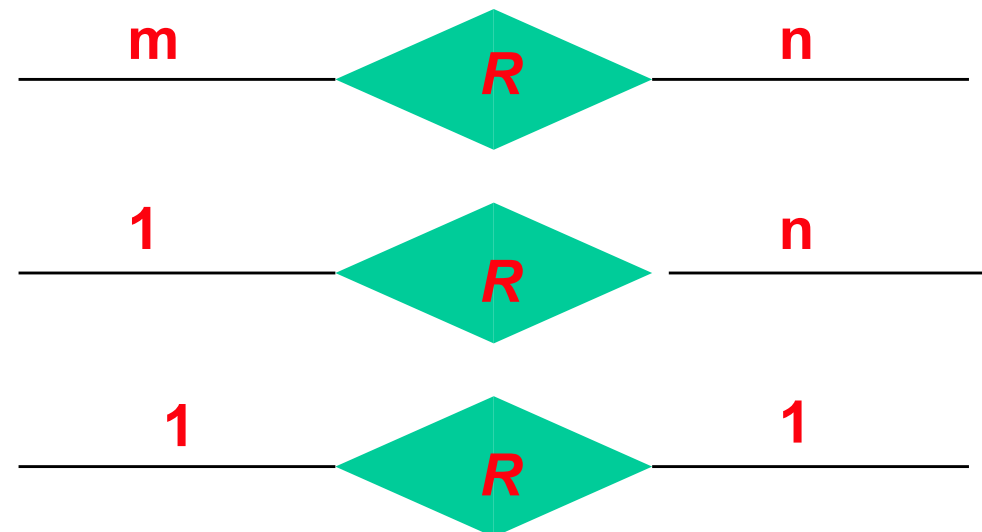
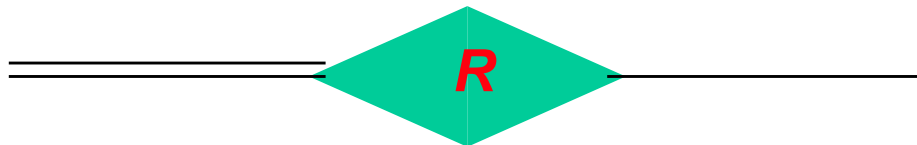
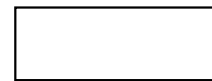
Many-to-many (M:N)

◆ Participation: whether every entity in the entity set participates in the relationship set: total v.s. partial

ER Model Diagram

■ ER Diagram

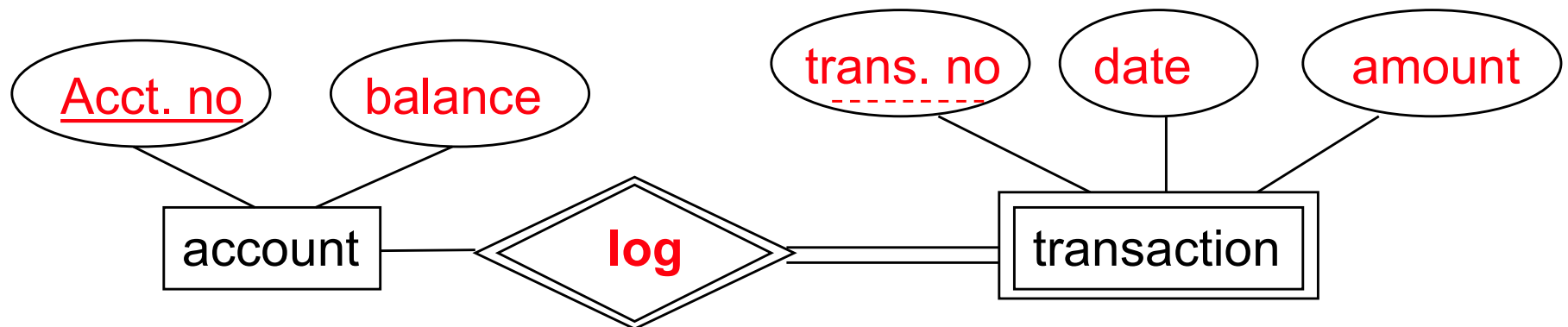
- ◆ Rectangles: Entity Sets
- ◆ Oval: Attributes
- ◆ Diamonds: Relationship Sets
- ◆ Lines: Attributes to Entity/Relationship Sets
or, Entity Sets to Relationship Sets



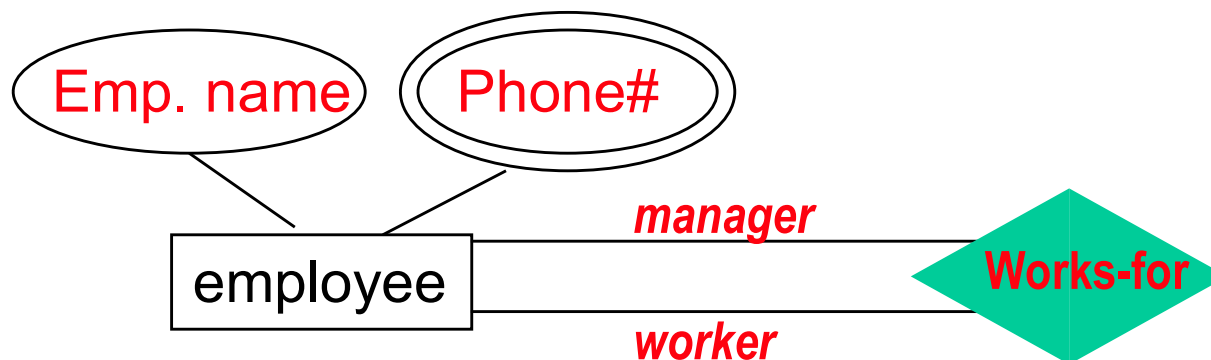
ER Model Diagram

■ Weak Entity Set

◆ an entity set that does NOT have enough attributes to form a key

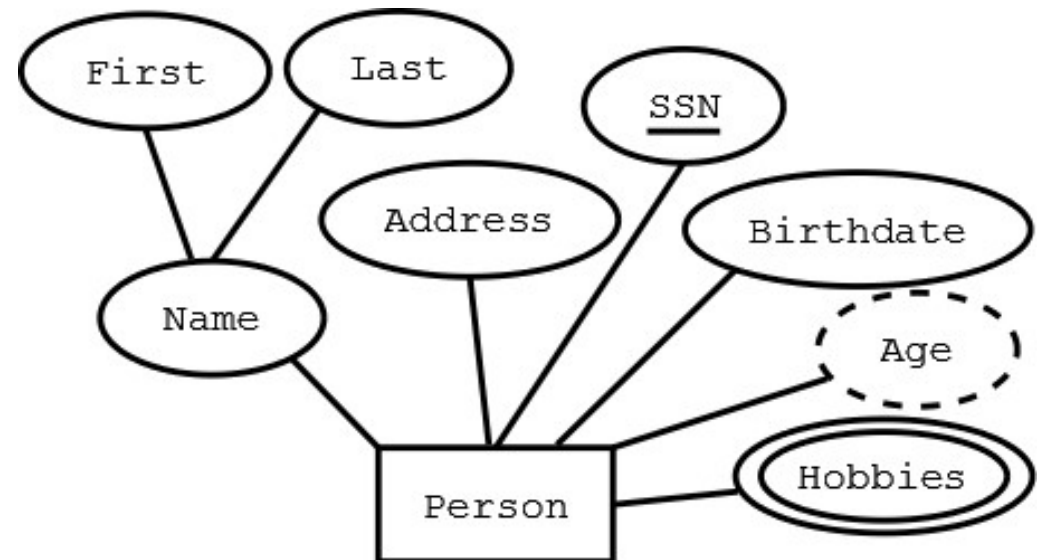
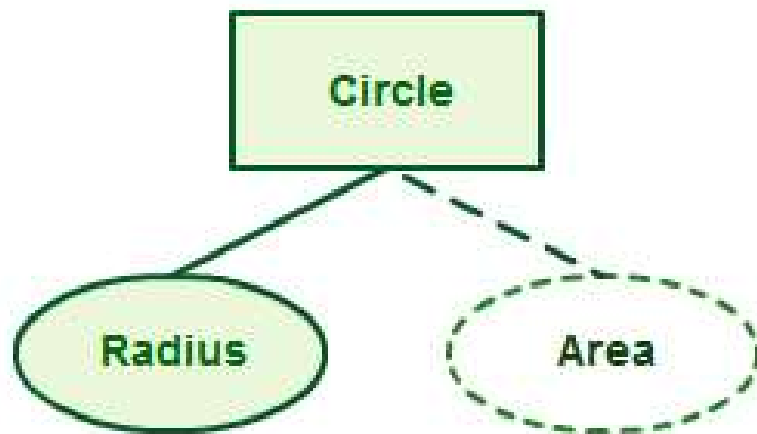


■ Role Indicators



ER Model Diagram

- **Derived Attribute:** An attribute which can be derived from other attributes is known as **derived attribute**.



Lecture 2:

ER model and relational model

Alternative (min, max) notation for relationship structural constraints:

■ Examples:

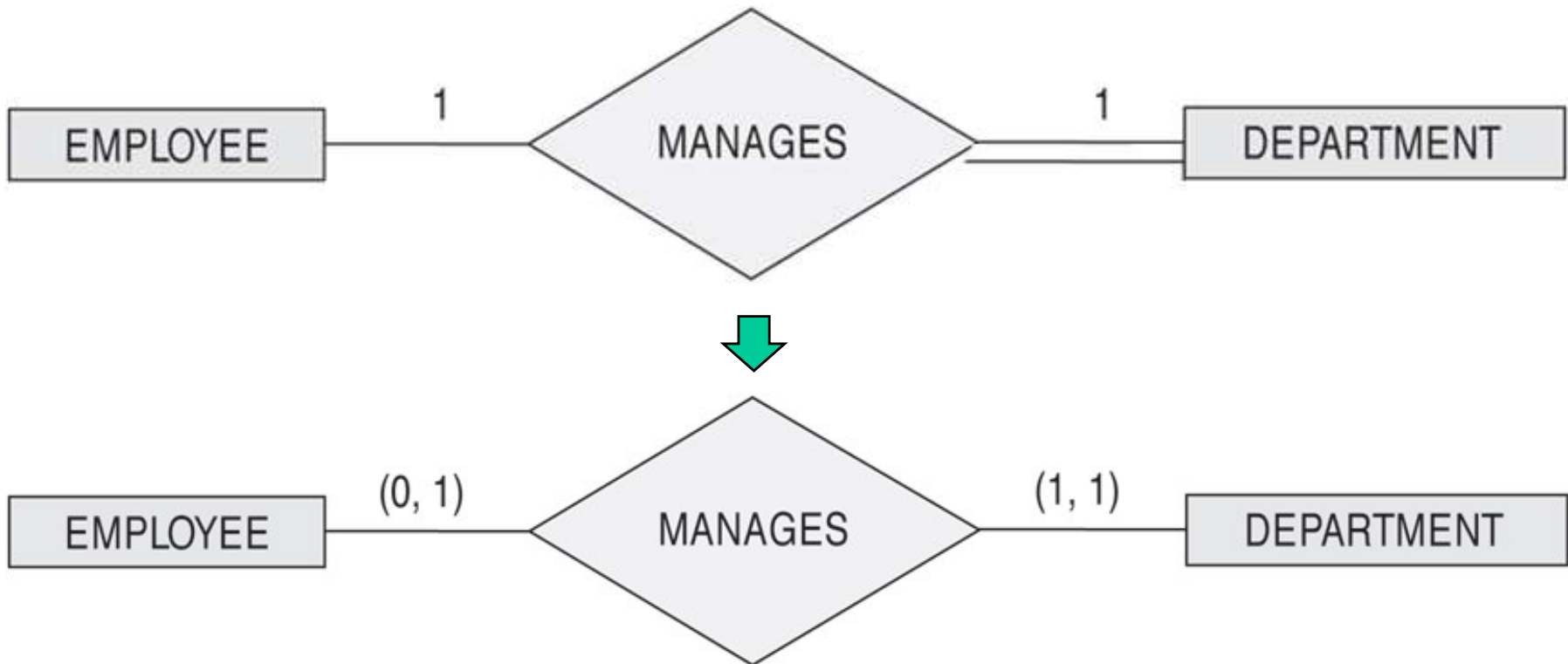
- ◆ A department has exactly one manager and an employee can manage at most one department
 - ◆ Specify (0,1) for participation of EMPLOYEE in MANAGES
 - ◆ Specify (1,1) for participation of DEPARTMENT in MANAGES

- ◆ An employee can work for exactly one department but a department can have any number (>1) of employees
 - ◆ Specify (1,1) for participation of EMPLOYEE in WORKS_FOR
 - ◆ Specify (1,n) for participation of DEPARTMENT in WORKS_FOR

The (min,max) notation for relationship constraints

One Employee may manage one Dept at most, zero Dept at least

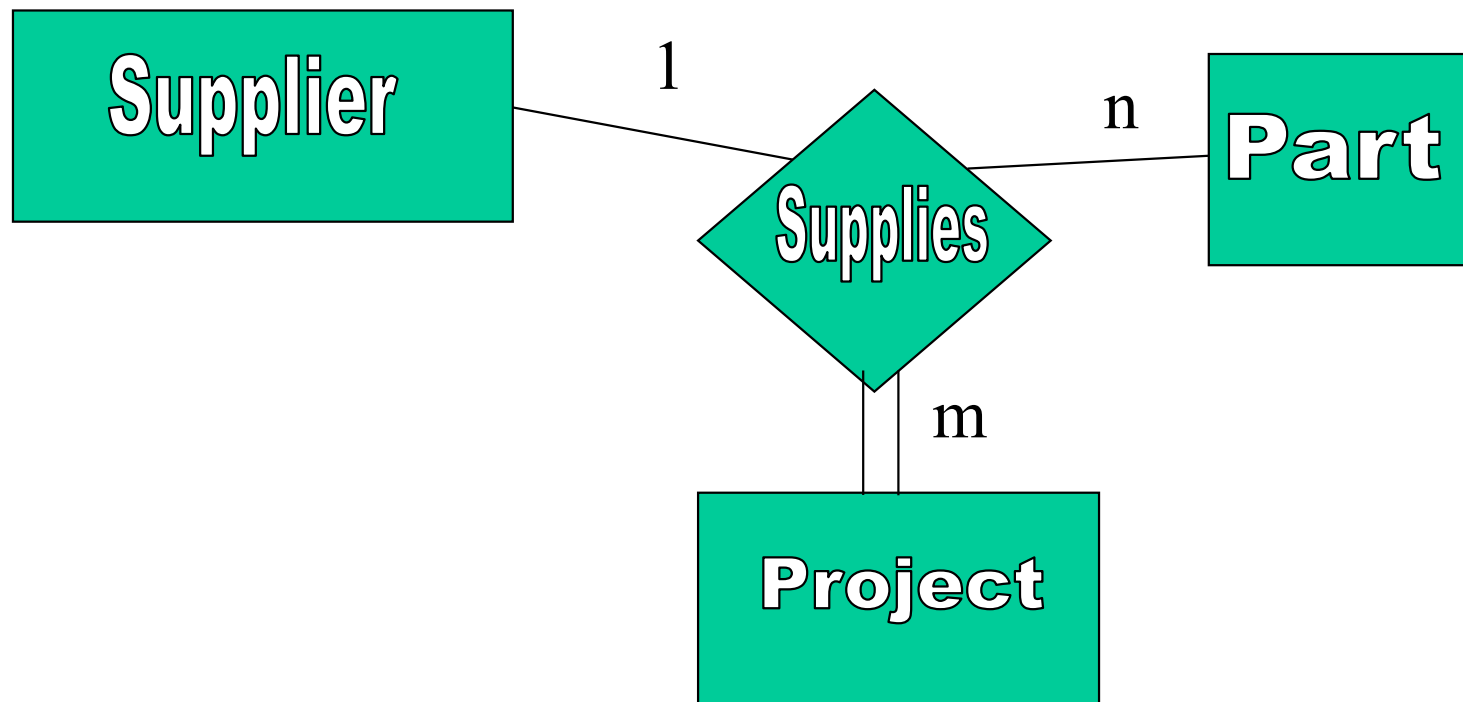
One Dept is managed by exactly one Employee



Relationships of Higher Degree

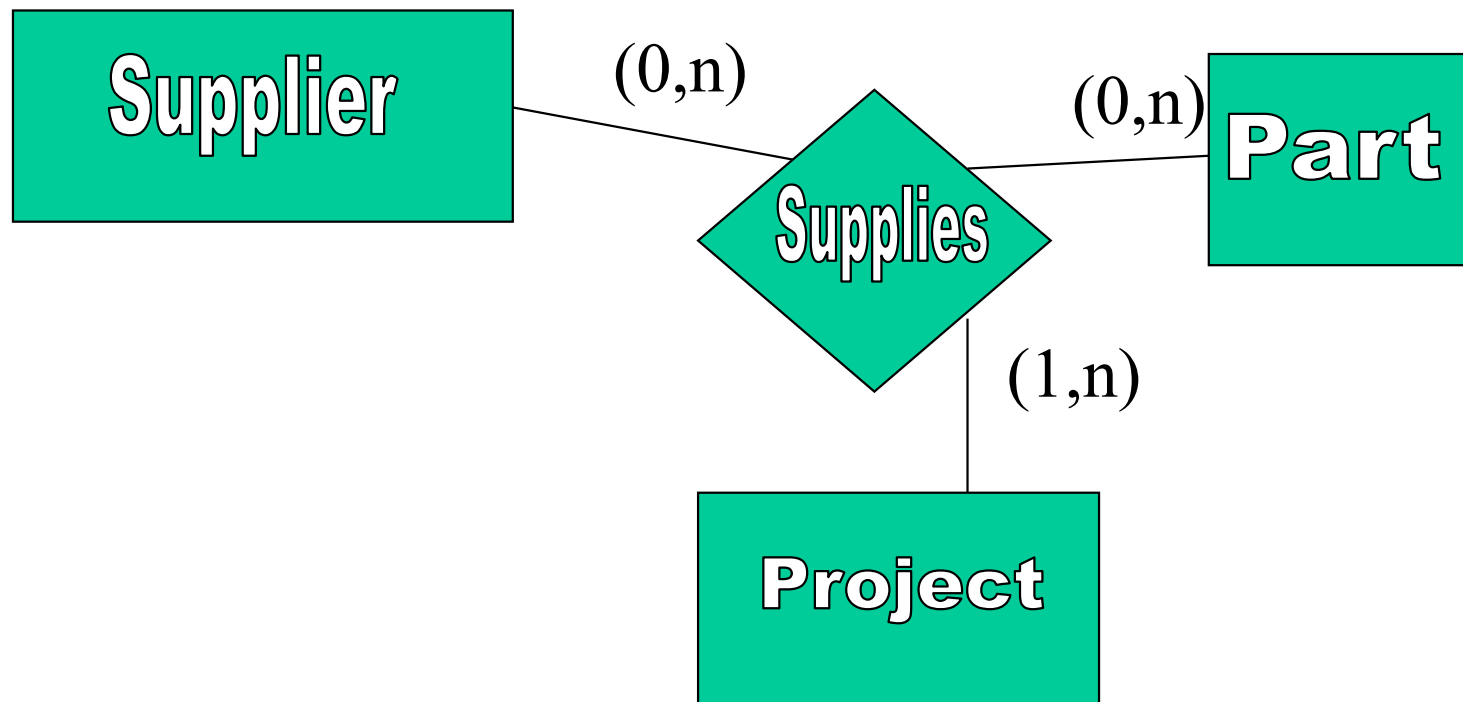
- Relationship types of degree 2 are called **binary**
- Relationship types of degree 3 are called **ternary** and of degree n are called n -ary
 - ◆ Supplier A supplies part B for project C
- In general, an **n -ary** relationship is not equivalent to n **binary** relationships
- **Constraints** are harder to specify for higher-degree relationships ($n > 2$) than for binary relationships

Constraints on Higher Order Relationship Types


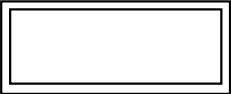

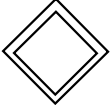



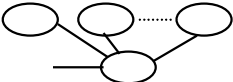

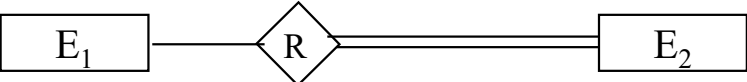
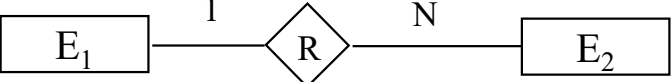
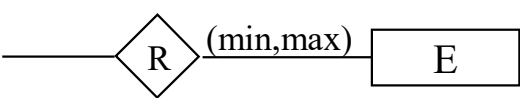


What does it mean to put 1:n:m on the three arms of the relationship?

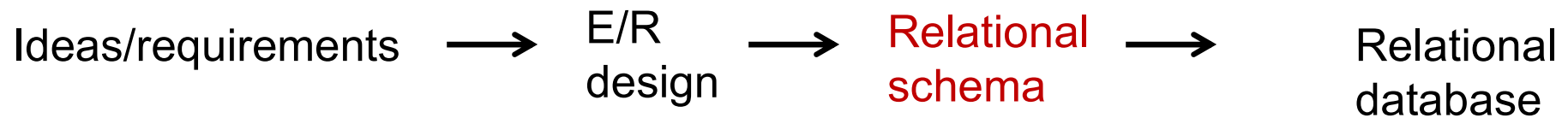
Constraints on Higher Order Relationship Types



Summary of ER-Diagram Notation

Symbol	Meaning
	ENTITY TYPE
	WEAK ENTITY TYPE
	RELATIONSHIP TYPE
	IDENTIFYING RELATIONSHIP TYPE
	ATTRIBUTE
	KEY ATTRIBUTE
	MULTIVALUED ATTRIBUTE
	COMPOSITE ATTRIBUTE
	DERIVED ATTRIBUTE
	TOTAL PARTICIPATION OF E ₂ IN R
	CARDINALITY RATIO 1:N FOR E ₁ :E ₂ IN R
	STRUCTURAL CONSTRAINT (min, max) ON PARTICIPATION OF E IN R

Database Modelling and Implementation



Example of a Relation

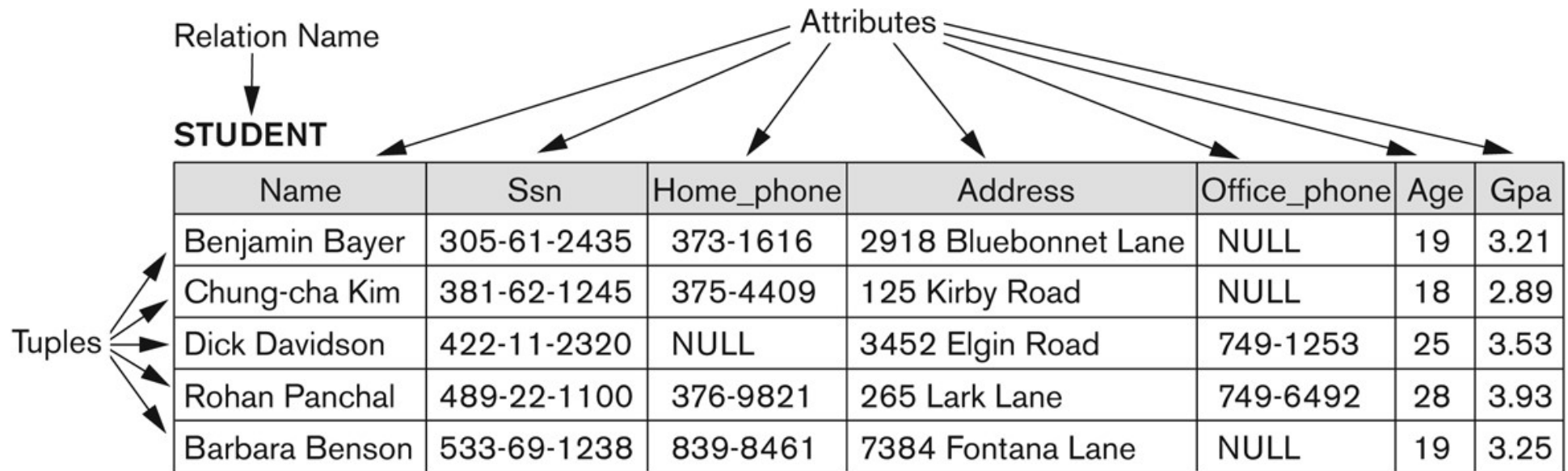


Figure 5.1

The attributes and tuples of a relation STUDENT.

STUDENT(Name:string, Ssn: integer, Home_phone: integer, Address:string, ...)

Definition Summary

<u>Informal Terms</u>	<u>Formal Terms</u>
Table	Relation
Column Header	Attribute
All possible Column Values	Domain
Row	Tuple
Table Definition	Schema of a Relation
Populated Table	State of the Relation

Characteristics Of Relations

- Ordering of tuples in a relation:
 - ◆ The tuples *are not considered to be ordered*, even though they appear to be in a tabular form (may have different *presentation* orders)

- Ordering of attributes in a relation schema R (and of values within each tuple):
 - ◆ We consider the attributes in $R(A_1, A_2, \dots, A_n)$ and the values in $t = \langle v_1, v_2, \dots, v_n \rangle$ to *be ordered*

ER->Relational Schema

■ **Translating ER diagram to relational model has the following steps:**

- 1 Map strong entity type into relation
- 2 Map weak entity + identifying relationship type into relation + foreign key
- 3 Map binary 1:1 relationship types into attributes (foreign key approach)
- 4 Map binary 1:N(N:1) Relationship types into attributes (foreign key approach)
- 5 Map binary M:N relationship type into relation + foreign key (cross reference approach)
- 6 Map N-ary relationship type into relation (cross reference approach)
- 7 Map multi-valued attribute into relation + foreign key

Example: ER of Company

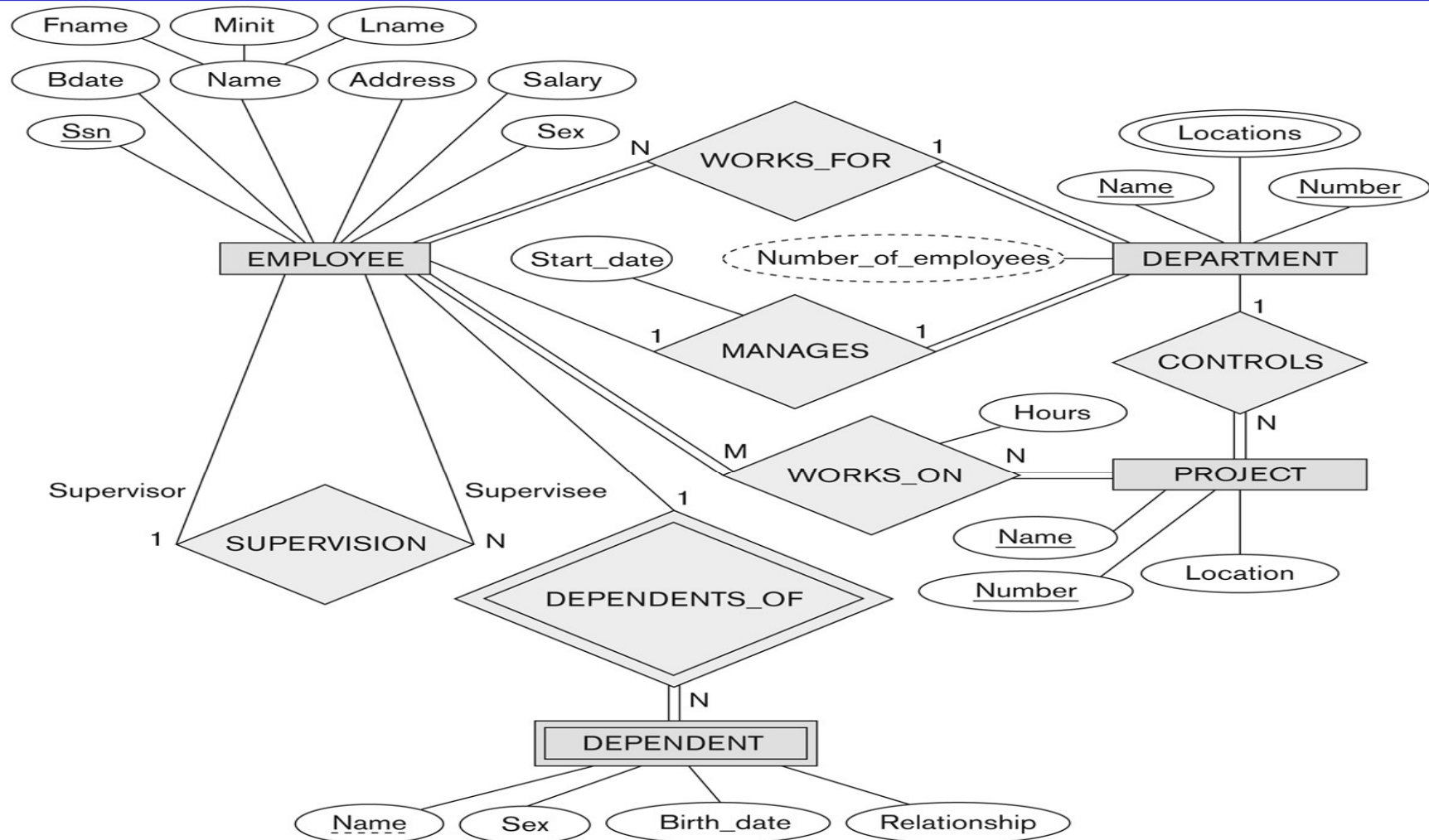


Figure 3.2

An ER schema diagram for the COMPANY database. The diagrammatic notation is introduced gradually throughout this chapter.

EMPLOYEE

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

DEPARTMENT

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

DEPT_LOCATIONS

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

PROJECT

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

WORKS_ON

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

DEPENDENT

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

Figure 9.2

Result of mapping the COMPANY ER schema into a relational database schema.

Discussion and Summary of Mapping for ER Model Constructs

Table 9.1 Correspondence between ER and Relational Models

ER MODEL	RELATIONAL MODEL
Entity type	<i>Entity</i> relation
1:1 or 1:N relationship type	Foreign key (or <i>relationship</i> relation)
M:N relationship type	<i>Relationship</i> relation and <i>two</i> foreign keys
<i>n</i> -ary relationship type	<i>Relationship</i> relation and <i>n</i> foreign keys
Simple attribute	Attribute
Composite attribute	Set of simple component attributes
Multivalued attribute	Relation and foreign key
Value set	Domain
Key attribute	Primary (or secondary) key

Lecture 3:

Integrity Constraints

Integrity Constraints

- A relational database schema is a set of relations $S = \{R_1, R_2, \dots, R_n\}$ and a set of integrity constraints IC
- Integrity Constraints determine which values are permissible and which are not in the database (table)
 - ◆ Constraints are conditions that must hold on all valid relation states
- Valid state Vs. invalid state
 - ◆ Invalid state: A database state that does not obey all the integrity constraints
 - ◆ Valid state: a state that satisfies all the constraints in the defined set of integrity constraints

Relational Integrity Constraints

- They are of three main types of constraints:
 - **Inherent or Implicit Constraints:** These are based on the data model itself. (E.g., relational model does not allow multiple values for any attribute)
 - **Schema-based or Explicit Constraints:** They are expressed in the schema by using the facilities provided by the model. (E.g., max. cardinality ratio constraint in the ER model)
 - **Application-based or Semantic constraints:** These are beyond the expressive power of the model and must be specified and enforced by the application programs. (e.g. the salary of an employee should not exceed the salary of the employee's supervisor)

Relational Integrity Constraints

- There are four *main types* of schema-based constraints that can be expressed in the relational model:
 - ◆ Domain constraints
 - ◆ Key constraints
 - ◆ Entity integrity constraints
 - ◆ Referential integrity constraints

Integrity Constraints

- In case of integrity violation, several actions can be taken:
 - ◆ cancel the operation that causes the violation
 - ◆ perform the operation but inform the user of the violation (e.g. ask the user to provide a valid value)
 - ◆ **trigger additional updates** so the violation is corrected (e.g. cascade the deletion by deleting tuples that reference the tuple being deleted)
 - ◆ execute a user-specified error-correction routine

Functional Dependency

■ Formal definition:

Let R be a relation schema, and $\alpha \subseteq R$, $\beta \subseteq R$ (ie, α and β are sets of R 's attributes). We say:

$$\alpha \rightarrow \beta$$

if in any legal relation instance $r(R)$, for **all** pairs of tuples $t1$ and $t2$ in r , we have:

$$(t1[\alpha] = t2[\alpha]) \Rightarrow (t1[\beta] = t2[\beta])$$

Inference Rules for FDs

IR1 (reflexive rule)	If Y is a subset of X , then $X \rightarrow Y$
IR2 (augmentation rule)	If $X \rightarrow Y$, then $XZ \rightarrow YZ$
IR3 (transitive rule)	If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$
IR4 (decomposition rule)	If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$
IR5 (union rule)	If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
IR6 (pseudotransitive rule)	If $X \rightarrow Y$ and $WY \rightarrow Z$, then $WX \rightarrow Z$

Closure of FDs

- **Closure** of a set F of **FDs** is the set F^+ of **all FDs** that can be inferred from F
- E.g., suppose we specify the following set F of obvious functional dependencies
 - ◆ $F = \{Ssn \rightarrow \{Ename, Bdate, Address, Dnumber\}, Dnumber \rightarrow \{Dname, Dmgr_ssn\}\}$
 - ◆ Then,
 - ◆ $Ssn \rightarrow \{Dname, Dmgr_ssn\}$
 - ◆ $Ssn \rightarrow Ssn$
 - ◆ $Dnumber \rightarrow Dname$
 - ◆
 - ◆ F^+ Including all FDs which can be inferred from F

Equivalence of Sets of FDs

- A set of functional dependencies F is said to **cover** another set of functional dependencies G if every FD in G is also in F^+ (G is a subset of F^+)
- Two sets of FDs F and G are **equivalent** if:
 - ◆ Every FD in F can be inferred from G , and
 - ◆ Every FD in G can be inferred from F
 - ◆ Hence, F and G are equivalent iff $F^+ = G^+$
- Example:
 - ◆ $F: A \rightarrow BC;$
 - ◆ $G: A \rightarrow B, A \rightarrow C$
 - ◆ $F^+ = G^+$

Closure of attribute set

- Closure of a set of attributes X with respect to F is the set X^+ of all attributes that are functionally determined by X
 - ◆ Note both X and X^+ are a set of attributes
- If X^+ consists of all attributes of R , X is a superkey for R
 - ◆ From the value of X , we can determine the values the whole tuple
- X^+ can be calculated by repeatedly applying IR1, IR2, IR3 using the FDs in F
- From X to find out X^+

Key Constraints

- Superkey of R:
 - ◆ A set of attributes that can uniquely identify a tuple is called a superkey.
 - ◆ It is a set of attributes SK, e.g., {A1, A2} of R with the following condition (Key constraint):
 - ◆ No two tuples in any valid relation state $r(R)$ will have the same value for SK
 - ◆ For any distinct tuples $t1$ and $t2$ in $r(R)$, $t1[SK] \neq t2[SK]$
- (Candidate) Key of R:
 - ◆ A "minimal" superkey
 - ◆ A key is a superkey K such that removal of any attribute from K results in a set of attributes that is not a superkey (does not possess the superkey uniqueness property)

Lecture 4: ***Normal Forms***

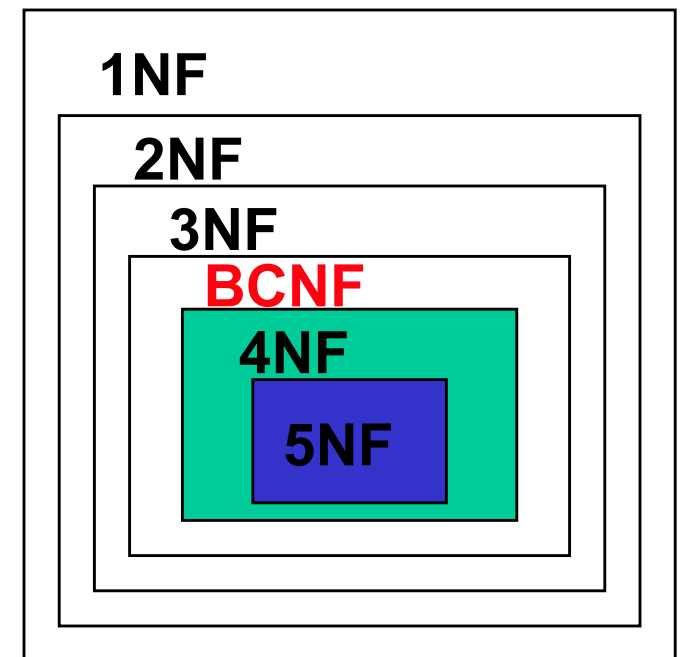
Relational Database Design

■ Introduction (con'd)

◆ *Normalization theory*

- based on **functional dependencies**

- * universe of relations
- * 1st Normal Form (1NF)
- * 2NF
- * 3NF
- * BCNF
- * ...



Definition with single candidate key

- 1st normal form
 - ◆ All attributes are simple (no composite or multivalued attributes)
- 2nd normal form
 - ◆ All non-prime attributes depend on the whole key (no partial dependency)
- 3rd normal form
 - ◆ All non-prime attributes **only** depend on the key (no transitive dependency)
- In general, 3NF is desirable and powerful enough
- But, still there are cases where a stronger normal form is needed

General Definitions of 2NF ,3NF,BCNF

Definition. A relation schema R is in **second normal form (2NF)** if every non-prime attribute A in R is not partially dependent on *any* key of R .¹¹

Definition. A relation schema R is in **third normal form (3NF)** if, whenever a *nontrivial* functional dependency $X \rightarrow A$ holds in R , either (a) X is a superkey of R , or (b) A is a prime attribute of R .

Definition. A relation schema R is in **BCNF** if whenever a *nontrivial* functional dependency $X \rightarrow A$ holds in R , then X is a superkey of R .

Test and Remedy for Normalization

Table 14.1 Summary of Normal Forms Based on Primary Keys and Corresponding Normalization

Normal Form	Test	Remedy (Normalization)
First (1NF)	Relation should have no multivalued attributes or nested relations.	Form new relations for each multivalued attribute or nested relation.
Second (2NF)	For relations where primary key contains multiple attributes, no nonkey attribute should be functionally dependent on a part of the primary key.	Decompose and set up a new relation for each partial key with its dependent attribute(s). Make sure to keep a relation with the original primary key and any attributes that are fully functionally dependent on it.
Third (3NF)	Relation should not have a nonkey attribute functionally determined by another nonkey attribute (or by a set of nonkey attributes). That is, there should be no transitive dependency of a nonkey attribute on the primary key.	Decompose and set up a relation that includes the nonkey attribute(s) that functionally determine(s) other nonkey attribute(s).

Lecture 5&6:

SQL: Structured Query Language

DB System Architecture

- *Data Query Language (DQL)*
 - a language used to make queries in databases
 - e.g. search records with giving conditions (sex="Female")
- *Data Manipulation Language (DML)*
 - a language that enables users to manipulate data
 - e.g. insert or delete records
- *Data Definition Language (DDL)*
 - a language for defining DB schema
 - e.g., create, modify, and remove database objects such as tables, indexes, and users.

DDL: Define database

CREATE/ALTER/DROP Table

- Create Database:

CREATE SCHEMA database_name AUTHORIZATION user-name;

- Create Table: CREATE TABLE table_name
(column_name1 data_type(size),
column_name2 data_type(size),....);

- Delete Table: DROP TABLE table-name;

- Update Table:

ALTER TABLE table-name
ADD Aj, Dj;

(to add new attribute Aj with
domain Dj to an existing table)

ALTER TABLE table-name
DROP Aj;

(to delete attribute Aj from
an existing table)

CREATE/DROP Index

```
CREATE [ UNIQUE ] INDEX <index name>  
ON <table name> ( <column name> [ <order> ] )  
[ CLUSTER ];
```

```
CREATE INDEX DnoIndex  
ON EMPLOYEE (Dno)  
CLUSTER;
```

- The optional keyword CLUSTER is used when the index to be created should also sort the data file records on the indexing attribute.
- The optional keyword UNIQUE is used when we constrain index field to be the key field.
- The value for <order> can be either ASC (ascending) or DESC (descending). The default is ASC.
- DROP INDEX** <index name>;

Views (Virtual Tables)

- Base table (base relation)
 - ◆ Relation and its tuples are **actually** (physically) created and stored as a **file** by the DBMS
 - ◆ The tables are stored in the secondary storage in the **specified format**
- Virtual table (view)
 - ◆ Single table **derived** from other base tables temporarily.
 - ◆ A view does not necessarily exist in physical form; it is just presented to the user through reconstruction (view) of base tables.

Views (Virtual Tables)

- CREATE VIEW command

- ◆ In V1, attributes retain the names from base tables.
In V2, attributes are assigned new names

```
V1:  CREATE VIEW  WORKS_ON1
      AS SELECT   Fname, Lname, Pname, Hours
          FROM     EMPLOYEE, PROJECT, WORKS_ON
          WHERE    Ssn=Essn AND Pno=Pnumber;
```

```
V2:  CREATE VIEW  DEPT_INFO(Dept_name, No_of_ems, Total_sal)
      AS SELECT   Dname, COUNT (*), SUM (Salary)
          FROM     DEPARTMENT, EMPLOYEE
          WHERE    Dnumber=Dno
          GROUP BY Dname;
```

New attribute
names



- DROP VIEW command

- ◆ DROP VIEW WORKS_ON1;

DML: Manipulate data

INSERT Command

- **INSERT** is used to add a single tuple (row) to a relation (table). We must specify the relation name and a list of values for the tuple.

```
U1:  INSERT INTO  EMPLOYEE  
      VALUES      ( 'Richard', 'K', 'Marini', '653298653', '1962-12-30', '98  
                    Oak Forest, Katy, TX', 'M', 37000, '653298653', 4 );
```

```
U3B:  INSERT INTO  WORKS_ON_INFO ( Emp_name, Proj_name,  
                                     Hours_per_week )  
      SELECT        E.Lname, P.Pname, W.Hours  
      FROM          PROJECT P, WORKS_ON W, EMPLOYEE E  
      WHERE         P.Pnumber=W.Pno AND W.Essn=E.Ssn;
```

The values for the attributes are obtained
from the results of the SELECT statement

DELETE Command

- **DELETE** command removes tuples from a relation.

DELETE FROM table-name;

◆ (Note: this operation only deletes all tuples from the table and the table is still there)

◆ Includes a **WHERE** clause to select the tuples to be delete

U4A: **DELETE FROM** EMPLOYEE
WHERE Lname='Brown';

U4B: **DELETE FROM** EMPLOYEE
WHERE Ssn='123456789';

U4C: **DELETE FROM** EMPLOYEE
WHERE Dno=5;

U4D: **DELETE FROM** EMPLOYEE;

UPDATE Command

- **UPDATE** command is used to modify attribute values of one or more selected tuples.
- Additional **SET** clause in the **UPDATE** command
 - ◆ Specifies attributes to be modified and new values

```
U5:    UPDATE    PROJECT
        SET      Plocation = 'Bellaire', Dnum = 5
        WHERE    Pnumber=10;
```

DQL: make query of data

Structure of SQL Queries

```
SELECT <attribute and function list>  
FROM <table list>  
[ WHERE <condition> ]  
[ GROUP BY <grouping attribute(s)> ]  
[ HAVING <group condition> ]  
[ ORDER BY <attribute list> ];
```

Tables as Sets in SQL

■ Set operations

- ◆ **UNION, INTERSECT, MINUS/EXCEPT** These set operations apply only to type compatible relations, so we must make sure that the two relations on which we apply the operation have the same attributes and that the attributes appear in the same order in both relations.

Query 4. Make a list of all project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.

Q4A: (SELECT
FROM
WHERE

DISTINCT Pnumber
PROJECT, DEPARTMENT, EMPLOYEE
Dnum=Dnumber **AND** Mgr_ssn=Ssn
AND Lname='Smith')

UNION
(SELECT
FROM
WHERE

DISTINCT Pnumber
PROJECT, WORKS_ON, EMPLOYEE
Pnumber=Pno **AND** Essn=Ssn
AND Lname='Smith');

Same attribute

The projects in the dept
with "Smith" as manager

The projects with
"Smith" as the worker

Nested Queries

- Nested queries

- ◆ Some queries require that existing values in the database be fetched and then used in a comparison condition.
- ◆ Such queries can be conveniently formulated by using nested queries, which are complete select-from-where blocks within another SQL query.
- ◆ The inner one is called nested query and the outer one is called the outer query.
- ◆ These nested queries can also appear in the **WHERE clause** or the FROM clause or the HAVING clause or other SQL clauses as needed.

Nested Queries with IN

■ Comparison operator IN

- ◆ Compares a single value v with a set (or multiset) of values V
- ◆ Evaluates “ v IN V ” to TRUE if v is one of the elements in V

e.g. Make a list of all project numbers for projects that involve an employee whose last name is ‘Smith’, either as a worker or as a manager of the department that controls the project.

```
Q4A:  SELECT DISTINCT Pnumber
        FROM PROJECT
        WHERE Pnumber IN
        ( SELECT Pnumber
          FROM PROJECT, DEPARTMENT, EMPLOYEE
          Dnum=Dnumber AND
          Mgr_ssn=Ssn AND Lname='Smith' )
        OR
        Pnumber IN
        ( SELECT Pno
          FROM WORKS_ON, EMPLOYEE
          Essn=Ssn AND Lname='Smith' );
```

The project no. of projects
that have an **manager**
with last name “Smith” →

The project no. of projects
that have an **employee**
with last name “Smith” →

Nested Queries, SOME/ANY, ALL

- In addition to the IN operator, a number of other comparison operators can be used to compare a single value v to a set or multiset V by combining the keywords ANY/SOME, ALL.
- ANY/SOME
 - ◆ $=$ ANY (or $=$ SOME) returns `TRUE` if the value v is equal to some value in the set V and is hence equivalent to `IN`
 - ◆ Other operators that can be combined with ANY (or SOME): $>$, $>=$, $<$, $<=$, and $<>$.
- ALL
 - ◆ ALL can also be combined with each of these operators: $=$, $>$, $>=$, $<$, $<=$, and $<>$.
 - ◆ E.g. $v > \text{ALL } V$ returns `TRUE` if the value v is greater than all the values in the set V .

Correlated Nested Queries

■ Correlated nested query

- ◆ Whenever a condition in the WHERE clause of a nested query references some attributes of a relation declared in the outer query.
- ◆ We can understand a correlated query better by considering that the nested query is evaluated once for each tuple in the outer query.

e.g. Retrieve the name of each employee who has a dependent with the same first name and the same sex as the employee.

```
SELECT E.Fname, E.Lname
FROM EMPLOYEE AS E
WHERE E.Ssn IN (SELECT D.Essn
                FROM DEPENDENT AS D
                WHERE E.Fname=D.Dependent_name AND E.Sex=D.Sex) ;
```

Correlated Nested Queries , EXISTS

■ **EXISTS/ NOT EXISTS function**

- ◆ Boolean functions that return TRUE or FALSE, used in a WHERE clause condition.
- ◆ Check whether the result of a nested query is empty or not
- ◆ EXISTS returns TRUE if not empty; NOT EXISTS returns TRUE if empty.
- ◆ Typically used in conjunction with a correlated nested query

e.g. Retrieve the name of each employee who has a dependent with the same first name and the same sex as the employee.

```
SELECT E.Fname, E.Lname
FROM EMPLOYEE AS E
WHERE EXISTS (SELECT *
              FROM DEPENDENT AS D
              WHERE E.Ssn=D.Essn AND E.Sex=D.Sex
              AND E.Fname=D.Dependent_name) ;
```

Summary of SQL Syntax

Table 7.2 Summary of SQL Syntax

```
CREATE TABLE <table name> ( <column name> <column type> [ <attribute constraint> ]  
                             { , <column name> <column type> [ <attribute constraint> ] }  
                             [ <table constraint> { , <table constraint> } ] )
```

```
DROP TABLE <table name>
```

```
ALTER TABLE <table name> ADD <column name> <column type>
```

```
SELECT [ DISTINCT ] <attribute list>
```

```
FROM ( <table name> { <alias> } | <joined table> ) { , ( <table name> { <alias> } | <joined table> ) }
```

```
[ WHERE <condition> ]
```

```
[ GROUP BY <grouping attributes> [ HAVING <group selection condition> ] ]
```

```
[ ORDER BY <column name> [ <order> ] { , <column name> [ <order> ] } ]
```

```
<attribute list> ::= ( * | ( <column name> | <function> ( ( [ DISTINCT ] <column name> | * ) ) )  
                    { , ( <column name> | <function> ( ( [ DISTINCT ] <column name> | * ) ) ) } ) )
```

```
<grouping attributes> ::= <column name> { , <column name> }
```

```
<order> ::= ( ASC | DESC )
```

```
INSERT INTO <table name> [ ( <column name> { , <column name> } ) ]
```

```
( VALUES ( <constant value> , { <constant value> } ) { , ( <constant value> { , <constant value> } ) } )
```

```
| <select statement> )
```

Summary of SQL Syntax

Table 7.2 Summary of SQL Syntax

DELETE FROM <table name>

[WHERE <selection condition>]

UPDATE <table name>

SET <column name> = <value expression> { , <column name> = <value expression> }

[WHERE <selection condition>]

CREATE [UNIQUE] INDEX <index name>

ON <table name> (<column name> [<order>] { , <column name> [<order>] })

[CLUSTER]

DROP INDEX <index name>

CREATE VIEW <view name> [(<column name> { , <column name> })]

AS <select statement>

DROP VIEW <view name>

NOTE: The commands for creating and dropping indexes are not part of standard SQL.

Lecture 7: ***Relational Algebra***

Relational Algebra

- Relational algebra: a formal language for the relational model
- The operations in relational algebra enable a user to specify basic retrieval requests (or queries)
- Relational algebra consists of a set of operations on relations to generate relations
- The result of an operation is a *new relation*
 - ◆ They can be further manipulated using operations
- A sequence of relational algebra operations forms a relational algebra expression

Relational Algebra Overview

- Relational algebra consists of several groups of operations
 - ◆ **Unary Relational Operations**
 - ◆ SELECT (symbol: σ (sigma))
 - ◆ PROJECT (symbol: π (pi))
 - ◆ RENAME (symbol: ρ (rho))
 - ◆ **Binary Relational Operations**
 - ◆ JOIN (several variations of JOIN exist)
 - ◆ DIVISION
 - ◆ Relational algebra Operations from **Set theory**
 - ◆ UNION (\cup), INTERSECTION (\cap), DIFFERENCE (or MINUS, $-$)
 - ◆ CARTESIAN PRODUCT (\times)
 - ◆ Additional Relational Operations
 - ◆ AGGREGATE FUNCTIONS (These compute summary of information: for example, SUM, COUNT, AVG, MIN, MAX)

Unary Relational Operations: SELECT

- The SELECT operation (denoted by σ (sigma)) is used to select a *subset* of the tuples from a relation based on a *selection condition*.
 - ◆ The selection condition acts as a *filter*
 - ◆ Keeps only those tuples that satisfy the *qualifying condition*
 - ◆ *Horizontal partitioning*
 - ◆ Tuples satisfying the condition are *selected* whereas the other tuples are discarded (*filtered out*)
- The general form of the *select* operation is:

$$\sigma_{\text{<condition>}}(R)$$

Unary Relational Operations: PROJECT

- PROJECT Operation is denoted by π (pi)
- This operation keeps certain **attributes** from a relation and discards the other attributes
 - ◆ PROJECT creates a **vertical partitioning**
 - ◆ The list of specified attributes is kept in each tuple
 - ◆ The other attributes in each tuple are discarded
- The general form of the *project* operation is:

$$\pi_{\langle \text{attribute list} \rangle}(R)$$

- ◆ π (pi) is the symbol used to represent the *project* operation
- ◆ $\langle \text{attribute list} \rangle$ is the desired list of attributes from relation R

Unary Relational Operations: RENAME

- The RENAME operator is denoted by ρ (rho)
- The general RENAME operation ρ can be expressed by any of the following forms:
 - ◆ $\rho_S(B_1, B_2, \dots, B_n)(R)$ changes both:
 - ◆ the relation name to **S**, *and*
 - ◆ the attribute names to B_1, B_1, \dots, B_n
 - ◆ $\rho_S(R)$ changes:
 - ◆ the *relation name* only to S
 - ◆ $\rho_{(B_1, B_2, \dots, B_n)}(R)$ changes:
 - ◆ the *attribute names* only to B_1, B_1, \dots, B_n

```
SELECT  E.Fname AS F_Name, E.Lname AS L_Name, E.Salary AS Salary
FROM    EMPLOYEE AS E
WHERE   E.Dno = 5
```

Operations from Set Theory

- UNION, denoted by \cup
 - ◆ The result of $R \cup S$, is a relation that includes all tuples that are either in R or in S or in both R and S

- INTERSECTION, denoted by \cap
 - ◆ The result of the operation $R \cap S$, is a relation that includes all tuples that are in both R and S

- DIFFERENCE (also called MINUS or EXCEPT), denoted by $-$
 - ◆ The result of $R - S$, is a relation that includes all tuples that are in R but not in S

Relational Algebra Operations from Set Theory: CARTESIAN PRODUCT

- CARTESIAN (or CROSS) PRODUCT Operation
 - ◆ This operation is used to combine tuples from two relations in a combinatorial fashion
 - ◆ Denoted by $R(A_1, A_2, \dots, A_n) \times S(B_1, B_2, \dots, B_m)$
 - ◆ Result is a relation Q with degree $n + m$ attributes:
 - ◆ $Q(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)$, in that order
 - ◆ The resulting relation state has one tuple for each combination of tuples - one from R and one from S
 - ◆ Hence, if R has n_R tuples (denoted as $|R| = n_R$), and S has n_S tuples, then $R \times S$ will have $n_R * n_S$ tuples
 - ◆ The two operands R and S do NOT have to be "type compatible"

Binary Relational Operations: JOIN

- JOIN Operation (denoted by \bowtie)
 - ◆ The sequence of CARTESIAN PRODECT followed by SELECT is used quite commonly to identify and select related tuples from two relations
 - ◆ A special operation, called JOIN combines this sequence into a single operation
 - ◆ The general form of a join operation on two relations $R(A_1, A_2, \dots, A_n)$ and $S(B_1, B_2, \dots, B_m)$ is:

$$R \bowtie_{\langle \text{join condition} \rangle} S$$

- ◆ R and S can be any relations that result from general *relational algebra expressions*
- ◆ R and S are not required to be type compatible.

NATURAL JOIN Operation

■ NATURAL JOIN Operation

◆ Another variation of JOIN called **NATURAL JOIN** — denoted by ***** was created to get rid of the **second (superfluous) attribute** in an EQUIJOIN condition

◆ because one of each pair of attributes with identical values is superfluous

◆ The standard definition of natural join requires that the two join attributes, or each pair of corresponding join attributes, *have the same name* in both relations.

e.g. $Q \leftarrow R(A,B,C,D) * S(C,D,E)$

◆ The implicit join condition includes *each pair* of attributes with the same name, “AND”ed together: $R.C=S.C$ AND $R.D=S.D$

◆ Result keeps only one attribute of each such pair:

◆ $Q(A,B,C,D,E)$

Binary Relational Operations: DIVISION

■ DIVISION Operation

- ◆ The division operation is applied to two relations
- ◆ $R(Z) \div S(X)$, where X is a subset of Z
- ◆ Let $Y = Z - X$ (and hence $Z = X \cup Y$); that is, let Y be the set of attributes of R that are not attributes of S
- ◆ The result of DIVISION is a relation $T(Y)$ that includes a tuple t if tuples t_R appear in R with $t_R[Y] = t$, and with
 - ◆ $t_R[X] = t_s$ for *every tuple* t_s in S
- ◆ For a tuple t to appear in the result T of the DIVISION, the values in t must appear in R in combination with *every tuple* in S .

Additional operators: Grouping and Aggregate Functions

- We can define an AGGREGATE FUNCTION operation, using the symbol \mathfrak{F} (pronounced script F) as follows:

$$\langle \text{grouping attributes} \rangle \mathfrak{F} \langle \text{function list} \rangle (R)$$

- $\langle \text{grouping attributes} \rangle$ is a list of attributes of the relation specified in R
- $\langle \text{function list} \rangle$ is a list of $\langle \text{function} \rangle$ (attribute) pairs. In each such pair, $\langle \text{function} \rangle$ is one of the allowed functions—such as SUM, AVERAGE, MAXIMUM, MINIMUM, COUNT
- E.g. retrieve each department number, the number of employees in the department, and their average salary

$$\text{Dno } \mathfrak{F} \text{ COUNT Ssn, AVERAGE Salary (EMPLOYEE)}$$

Lecture 8

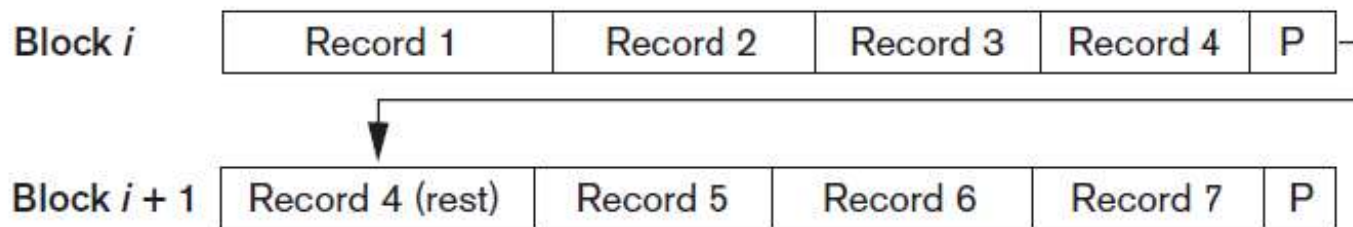
Files and Hashed Files

Files of Records

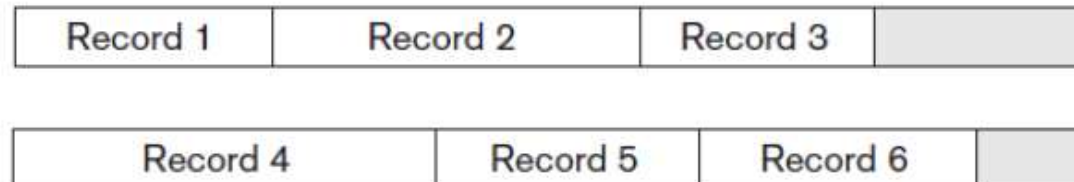
- When the block size is larger than the record size, each block will contain numerous records.
- **Blocking**: Refers to storing a number of records into one block on the disk
- **Blocking factor (bfr)** refers to the number of records per block
- Suppose that the block size is B bytes. For a file of fixed-length records of size R bytes, with $B \geq R$, we can fit $\text{bfr} = \lfloor B/R \rfloor$ records per block, where the $\lfloor x \rfloor$ (floor function) rounds down the number x to an integer.
- There may be empty space in a block if an integral number of records do not fit into one block: $B - \lfloor B/R \rfloor * R$

Files of Records

- File records can be **unspanned** or **spanned**
 - ◆ Spanned: a record can be stored in more than one block



- ◆ Unspanned: no record can span two blocks



- In a file of fixed-length records, unspanned blocking is usually used.
- For variable-length records, either a spanned or an unspanned organization can be used. it is advantageous to use spanning to reduce the lost space in each block.

Unordered Files

- Also called a **heap** file (records are unordered)
- New records are **inserted at the end** of the file
 - ◆ Arranged in their insertion sequence
 - ◆ record insertion is efficient (add to the end)
- A **linear search** through the file records is necessary to search for a record
 - ◆ For a file of b blocks, this requires reading and searching half the file blocks on **average**, and is hence quite expensive ($b/2$)

◆ Word

9	16	50	2	10	4	8	12	60	100
---	----	----	---	----	---	---	----	----	-----

Ordered Files

- Also called a **sequential** file (records are ordered)
- File records are kept sorted by the values of an *ordering field*
- Reading the records in **order of the ordering field** is quite efficient
- A **binary search** can be used to search for a record on its *ordering field* value
 - ◆ This requires reading and searching $\log_2 b$ of the file blocks on the average, an improvement over linear search
- **Insertion is expensive**: records must be inserted in the correct order
 - ◆ It is common to keep a separate **unordered overflow** file for new records to improve insertion efficiency; this is

2	4	8	9	10	12	16	50	60	100
---	---	---	---	----	----	----	----	----	-----

 ordered file.

Hashed Files

- Hashing for disk files is called External Hashing (files on disk)
- The file blocks are divided into M equal-sized buckets, numbered $\text{bucket}_0, \text{bucket}_1, \dots, \text{bucket}_{M-1}$. A bucket is either one disk block or a cluster of contiguous disk blocks.
- One of the file fields is designated to be the hash key of the file
- Suppose there is a hash function $h(K)$ that takes a hash key K as an input to compute an integer in the range 0 to $B - 1$ where B is the number of buckets
 - ◆ $h(K) \Rightarrow 0 \text{ to } B - 1$
- A table maintained in the file header converts the bucket number into the corresponding disk block address.

Hashed Files – Collision Resolution

- There are numerous methods for collision resolution:
 - ◆ **Open addressing**: proceeding from the occupied position specified by the hash address, the program checks the subsequent positions in order until an unused (empty) position is found.
 - ◆ Linear Probing
 - ◆ If collide, try $\text{Bucket_id}+1$, $\text{Bucket_id}+2$, ..., $\text{Bucket_id}+n$
 - ◆ Quadratic Probing
 - ◆ If collide, try $\text{Bucket_id}+1$, $\text{Bucket_id}+4$, ..., $\text{Bucket_id}+n^2$
 - ◆ What are the differences in performance?

Hashed Files Collision Resolution

■ Chaining:

- ◆ For this method, various overflow locations are kept, usually by extending the array with a number of overflow positions
- ◆ In addition, a **pointer field** is added to each bucket
- ◆ A collision is resolved by placing the new record in an unused overflow bucket and setting the pointer of the occupied hash address bucket to the address of that overflow bucket

■ Multiple hashing:

- ◆ The program applies a second hash function if the first results in a collision
- ◆ If another collision results, the program uses open addressing or applies a third hash function and then uses open addressing if necessary

Extendible and Dynamic Hashing

- Dynamic and Extendible Hashing Techniques
 - ◆ Hashing techniques are extended to allow dynamic growth and shrinking of the number of buckets.
 - ◆ These techniques include the following: dynamic hashing and extendible hashing
- Both dynamic and extendible hashing use an access structure to represent hash function $h(K)$, which is called directory
 - ◆ In dynamic hashing the directory is a binary tree
 - ◆ In extendible hashing the directory is an array

Extendible Hashing

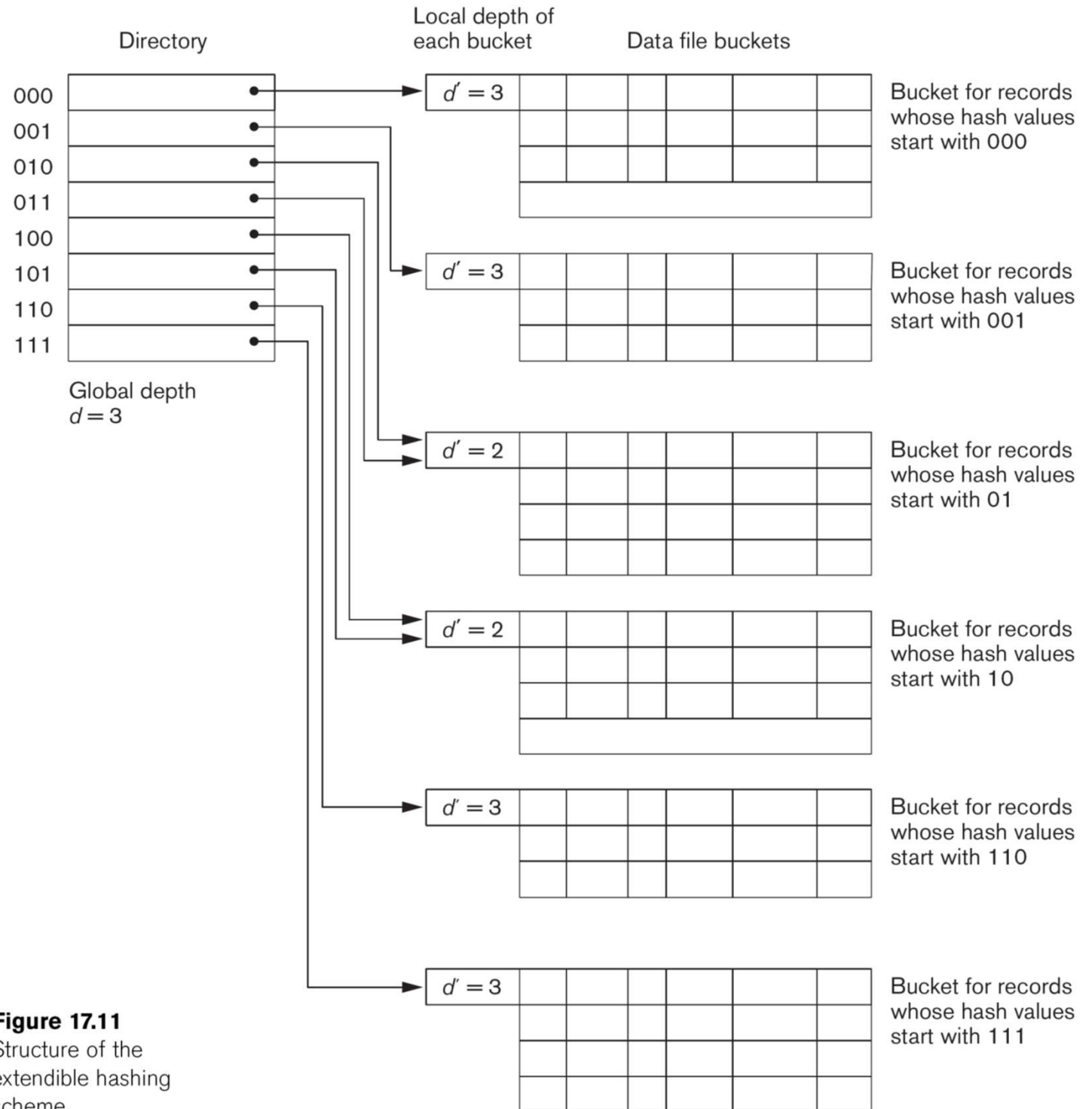


Figure 17.11
Structure of the
extendible hashing
scheme.

Extendible Hashing

- A directory consisting of an array of 2^d bucket addresses is maintained
- d is called the **global depth** of the directory
- The integer value corresponding to the **first (high-order) d** bits of a hash value is used as an index to the array to determine a directory entry and the address of the bucket storing the records
- A **local depth d'** stored with each bucket specifies the number of bits on which the bucket contents are based.
- The value of d' can be increased or decreased by one at a time to handle overflow or underflow respectively

Dynamic Hashing

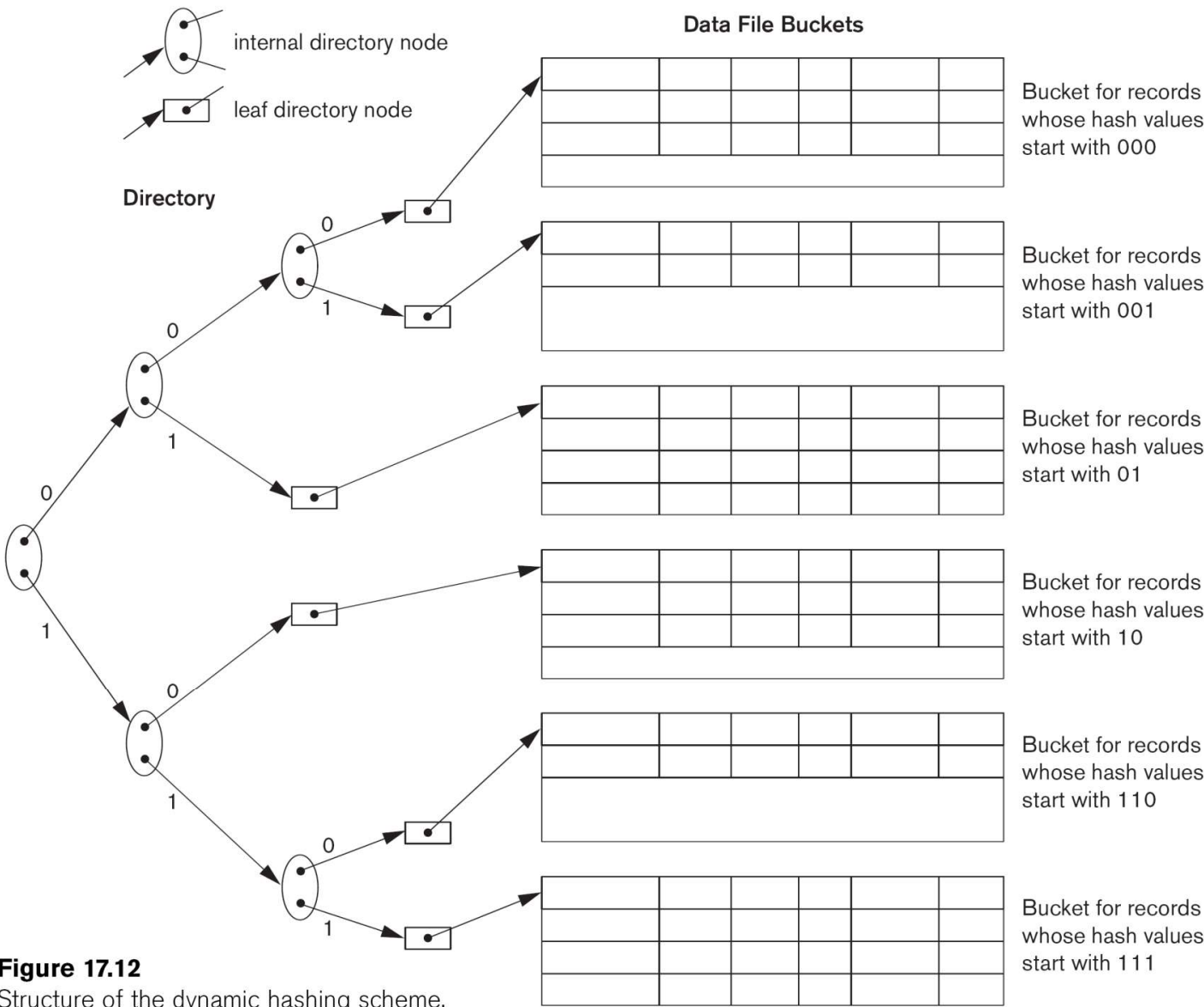


Figure 17.12
Structure of the dynamic hashing scheme.

Lecture 9

Indexing Techniques

Types of Single Level Indexes

- Single level index: index entry directly points to data record
- Primary index (ordering key field)
 - ◆ Specified on the ordering key field of an ordered file of records
- Cluster index (ordering non-key field)
 - ◆ The ordering field of the index is not a key field and numerous records in the data file can have the same value for the ordering field (repeating value attributes)
- Secondary index (non-ordering field)
 - ◆ The index field is specified on any non-ordering field of a data file
- Dense or sparse indexes
 - ◆ A dense index has an index entry for every record in the data file. Thus larger index size
 - ◆ A sparse index, on the other hand, has index entries for only some records. Thus smaller index size

Summary of single-level index

Table 17.1 Types of Indexes Based on the Properties of the Indexing Field

	Index Field Used for Physical Ordering of the File	Index Field Not Used for Physical Ordering of the File
Indexing field is key	Primary index	Secondary index (Key)
Indexing field is nonkey	Clustering index	Secondary index (NonKey)

Table 17.2 Properties of Index Types

Type of Index	Number of (First-Level) Index Entries	Dense or Nondense (Sparse)	Block Anchoring on the Data File
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no ^a
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records ^b or number of distinct index field values ^c	Dense	No

^aYes if every distinct value of the ordering field starts a new block; no otherwise.

^bFor option 1.

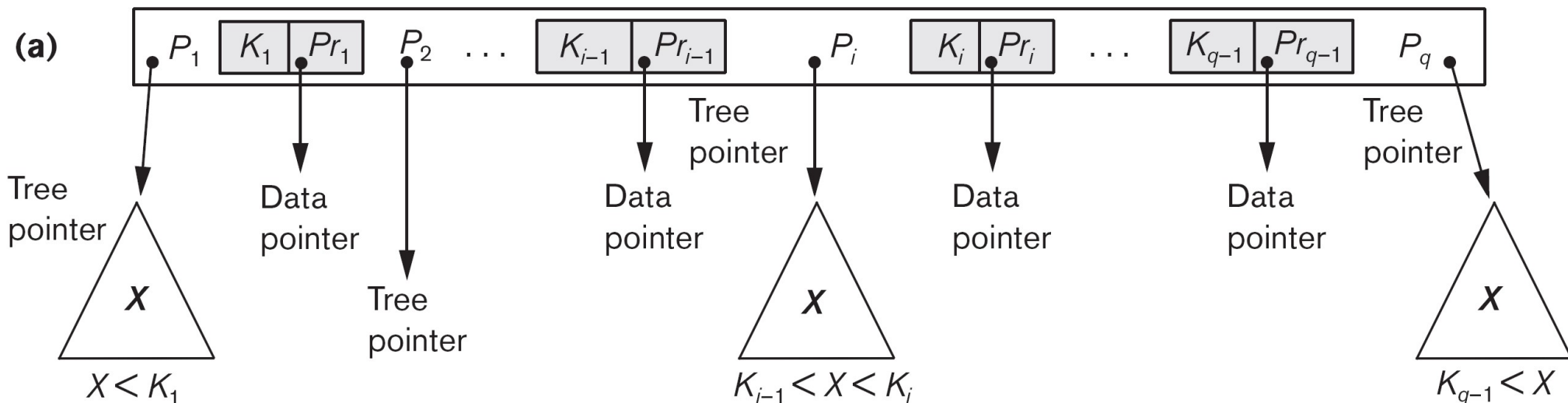
^cFor options 2 and 3.

Multi-Level Indexes

- Because a single-level index is an **ordered file**, we can create a **primary index** to the index itself
 - ◆ In this case, the original index file is called the **first-level index** and the index to the index is called the **second-level index**
- We can repeat the process, creating a third, fourth, ..., top level until all entries of the **top level** fit in one disk block
- A **multi-level index** can be created for any type of single-level index (primary, secondary, clustering) as long as the single-level index consists of *more than one* disk block

B-tree Index

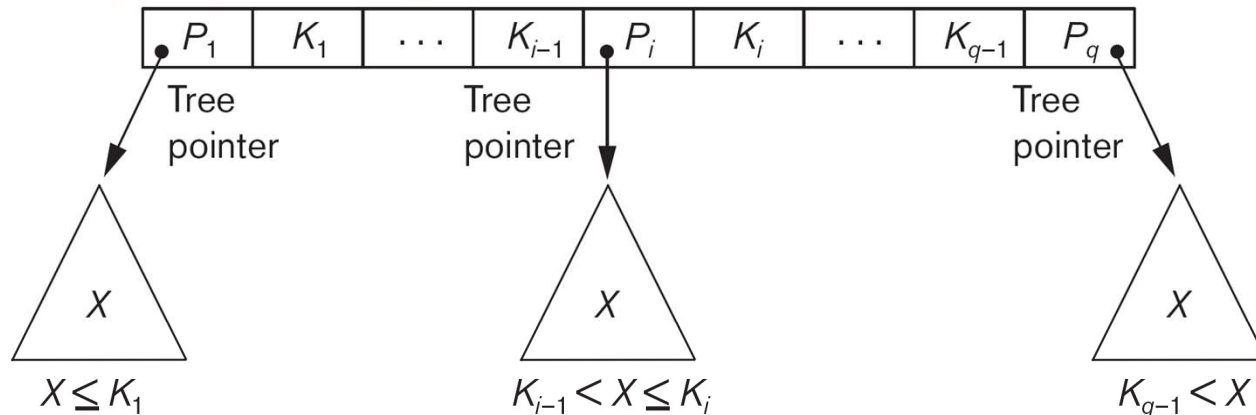
- B-tree organizes its nodes into a tree
- It is **balanced (B)** as all paths from the root to a leaf node have the **same length**
- Each internal node in a B-tree is of the form $\langle P_1, \langle K_1, Pr_1 \rangle, P_2, \langle K_2, Pr_2 \rangle, \dots, \langle K_{q-1}, Pr_{q-1} \rangle, P_q \rangle$
- Each P_i is a **tree pointer** to another node in the next level of B-tree
- Each Pr_i is a **data pointer** points to the record whose search key field value is equal to K_i



B⁺-tree Index

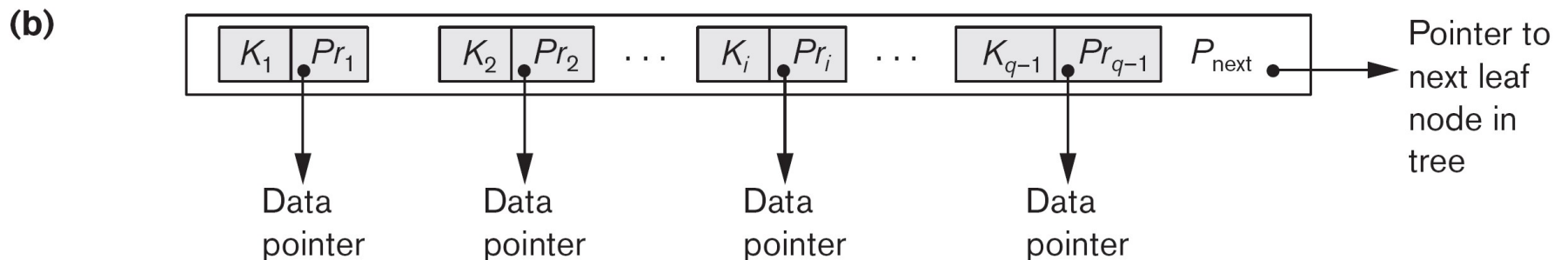
- The structure of internal nodes of a B⁺-tree
 - ◆ Each internal node in a B⁺-tree is of the form $\langle P_1, K_1, P_2, K_2, \dots, P_{q-1}, K_{q-1}, P_q \rangle$ and P_i is a tree pointer.
 - ◆ Within each node, $K_1 < K_2 < \dots < K_{q-1}$
 - ◆ For all search key field values X in the subtree pointed by P_i , we have $K_{i-1} < X \leq K_i$ for $1 < i < q$; $X \leq K_1$ for $i = 1$; and $K_{q-1} < X$ for $i = q$
 - ◆ Each internal node has at most q tree pointers, $q-1$ keys. q is the order of the tree. (full)
 - ◆ Each internal node except root node has at least $\lceil q/2 \rceil$ tree pointers $\lceil q/2 \rceil - 1$ keys. (half-full)

(a)



B⁺-tree Index

- The structure of the leaf nodes of a B⁺-tree
 - ◆ Each leaf node is of the form
 - ◆ $\langle \langle K_1, Pr_1 \rangle, \langle K_2, Pr_2 \rangle, \dots, \langle K_{q-1}, Pr_{q-1} \rangle, P_{next} \rangle$ and Pr_i is a data pointer and P_{next} points to the next leaf node of the tree
 - ◆ Within each node, $K_1 < K_2 < \dots < K_{q-1}$
 - ◆ Each leaf node has at most q pointers, $q-1$ keys (full), and at least $\lceil q/2 \rceil$ pointers, $\lceil q/2 \rceil - 1$ keys (half-full), q is the order of leaf node.
 - ◆ All leaf nodes are at the same level



B⁺-tree Index Insertion

- Step1: Descend to the leaf node where the key fits

- Step 2:
 - ◆ (Case 1): If the node has an empty space, insert the key into the node.
 - ◆ (Case 2) If the node is already **full**, split it into two nodes by the middle key value, distributing the keys evenly between the two nodes, so each node is **half full**.
 - ◆ (Case 2a for leaf node) If the node is a leaf, take a copy of the middle key value, and repeat step 2 to insert it into the parent node.
 - ◆ (Case 2b for internal node) If the node is a non-leaf, exclude the middle key value during the split and repeat step 2 to insert this excluded value into the parent node.

B⁺-tree Index Deletion

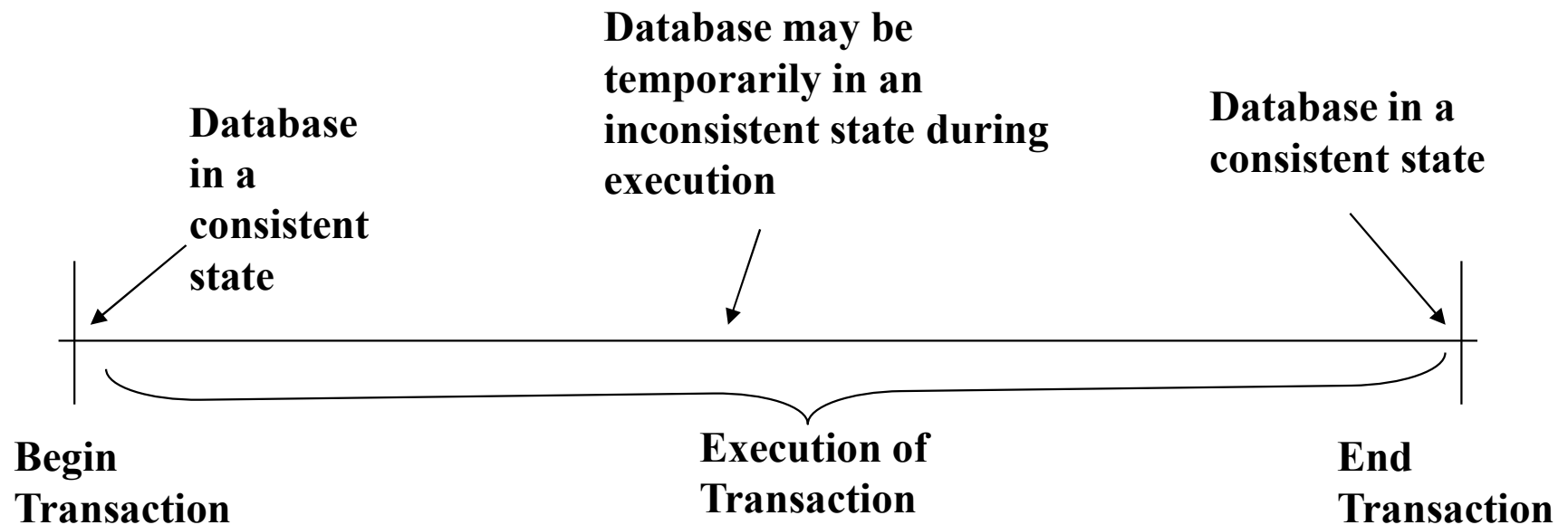
- Step1: Descend to the leaf node where the key fits
- Step 2: Remove the required key and associated reference from the node.
 - ◆ (Case 1): If the node still has **half-full** keys, repair the keys in parent node to reflect the change in child node if necessary and stop.
 - ◆ (Case 2): If the node is **less than half-full**, but left or right sibling node is more than half-full, redistribute the keys between this node and its sibling. Repair the keys in the level above to represent that these nodes now have a different “split point” between them.
 - ◆ (Case 3): If the node is **less than half-full**, and left and right sibling node are just half-full, merge the node with its sibling. Repeat step 2 to delete the unnecessary key in its parent.

Lecture 10:

Transaction Management

Transaction Structure & Database Consistency

Example: T1: Begin; R(a); R(b) $c = a + b$; W(c); End



The whole transaction is considered as an **atomic unit**

- ◆ **Partial results are not allowed and is considered to be incorrect**
- ◆ **Atomicity: All or nothing**

Transaction Schedule

- Transaction schedule
 - ◆ When transactions are executing **concurrently** in an **interleaved** fashion or **serially**, the order of execution of **operations** from all transactions forms a **transaction schedule**
- A schedule S of n transactions T_1, T_2, \dots, T_n is an ordering of the operations of the transactions subject to the constraint:
 - ◆ For each transaction T_i that participates in S , the operations of T_i in S must appear in the same order as in T_i (operations from other transactions T_j can be interleaved with the operations of T_i in S)
- A concurrent schedule: a new transaction starts **BEFORE** the completion of the current transaction
- A serial schedule: a new transaction only starts **AFTER** the current transaction is completed

Schedules Classified on Serializability

- Serial schedule:

- ◆ A schedule S is **serial** if, for every transaction T participating in the schedule, all the operations of T are executed **consecutively** in the schedule
- ◆ Serial schedules can maintain the database consistency
 - ◆ BUT, poor performance

- Serializable schedule:

- ◆ A concurrent schedule S which is conflict equivalent to a **serial** schedule.
- ◆ Can guarantee the database consistency and can have better performance.

Serialization Graphs

- The determination of a **conflict serializable schedule** can be done by the use of **serialization graph (SG)** or called **precedence graph**
- A serialization graph tells the **effective** execution order of a set of transactions
- The set of edges consists of all edges $T_i \rightarrow T_j$ for which one of the following three conditions holds:
 - ◆ W/R conflict: T_i executes write(x) before T_j executes read(x)
 - ◆ R/W conflict: T_i executes read(x) before T_j executes write(x)
 - ◆ W/W conflict: T_i executes write(x) before T_j executes write(x)
- Each edge $T_i \rightarrow T_j$ in a SG means that at least one of T_i 's operations precede and conflict with one of T_j 's operations
- Serializability theorem:
 - ◆ A schedule is **serializable** iff the SG is **acyclic**

Schedules Classified on Recoverability

- Non-Recoverable schedule:

- ◆ The committed transaction with dirty-read problem cannot be rolled back.

- Recoverable schedule:

- ◆ No committed transaction needs to be rolled back.

- ◆ A schedule S is recoverable if

For all T_i and T_j where T_i read an item written by T_j ,
 T_j commits before T_i

Schedules Classified on Recoverability

- Cascadeless schedule:

- ◆ Every transaction reads only the items that are written by committed transactions.

- ◆ In other words

- Before T_i reads an item written by T_j , T_j is already committed

- Strict Schedules:

- ◆ A schedule in which a transaction can neither read or write an item X until the last transaction that wrote X has committed.

ACID Properties of Transactions

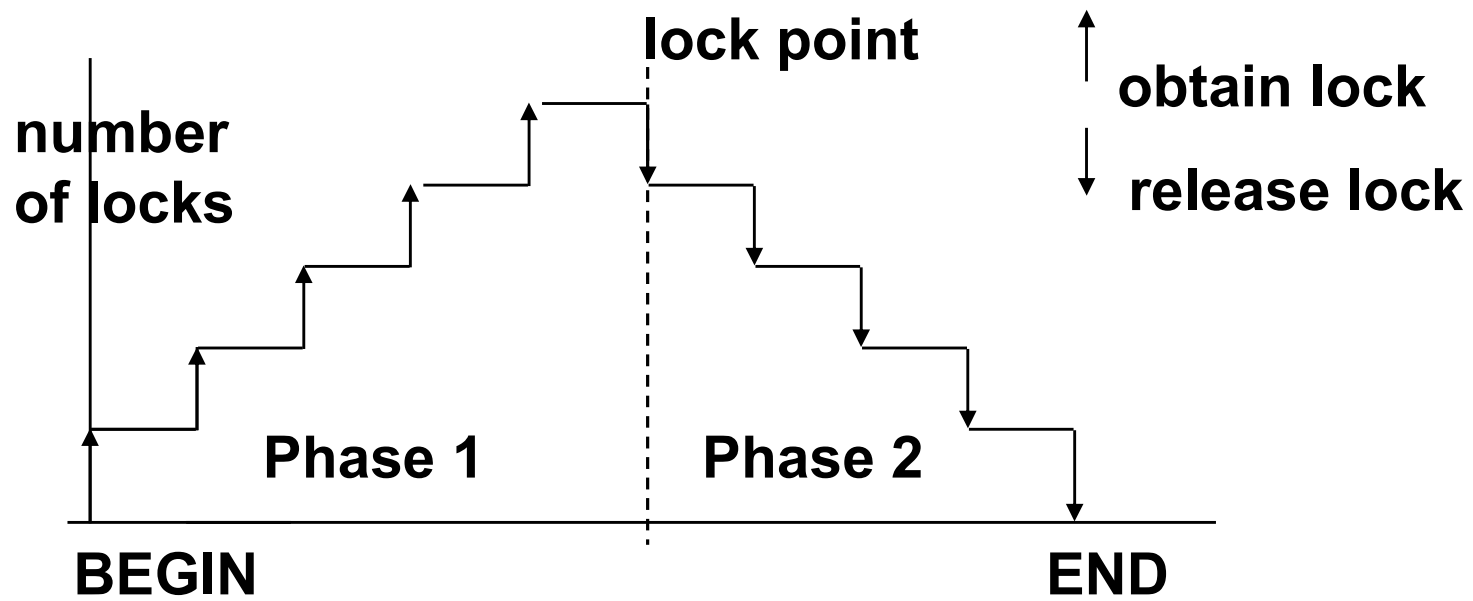
- **Atomicity:** A transaction is an atomic unit of processing. It is either performed completely or not performed at all (**all or nothing**)
- **Consistency:** A correct execution of a transaction must take the database from one consistent state to another (**correctness**)
- **Isolation:** A transaction should not make its updates visible to other transactions until it is committed (**no partial results**)
- **Durability:** Once a transaction changes the database state and the changes are committed, these changes must never be lost because of subsequent failure (**committed and permanent results**)

Lecture 11:

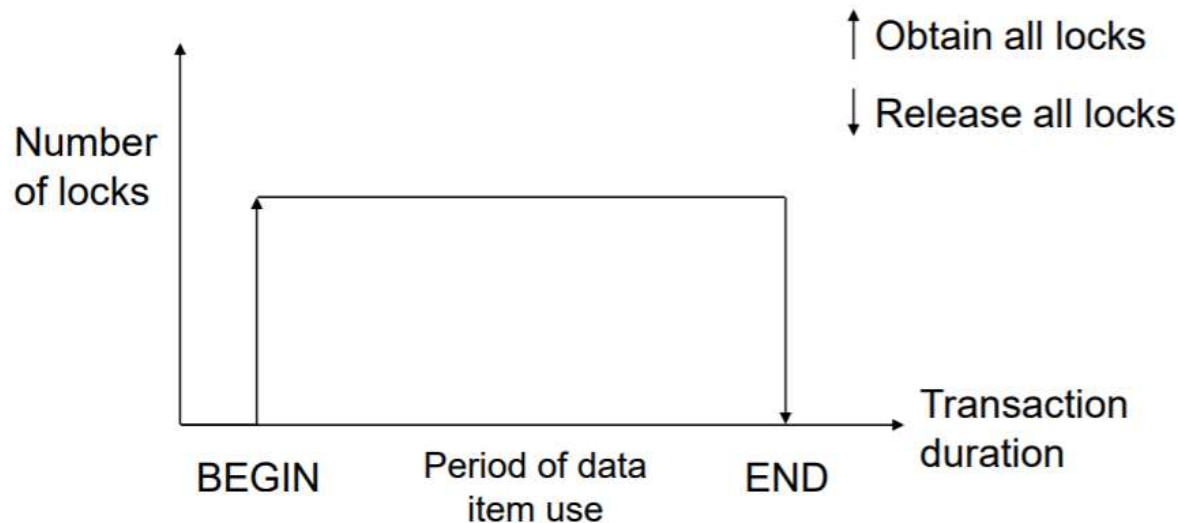
Concurrency Control

Basic Two Phase Locking (B2PL)

- The two phase rule: growing phase and shrinking phase
- It can guarantee that all pairs of conflicting operations of two transactions are scheduled in the same order, Why?
 - E.g., $T1 \rightarrow T2$ or $T2 \rightarrow T1$ and NO $T1 \leftrightarrow T2$
- B2PL enforce serializability but may produce deadlock.

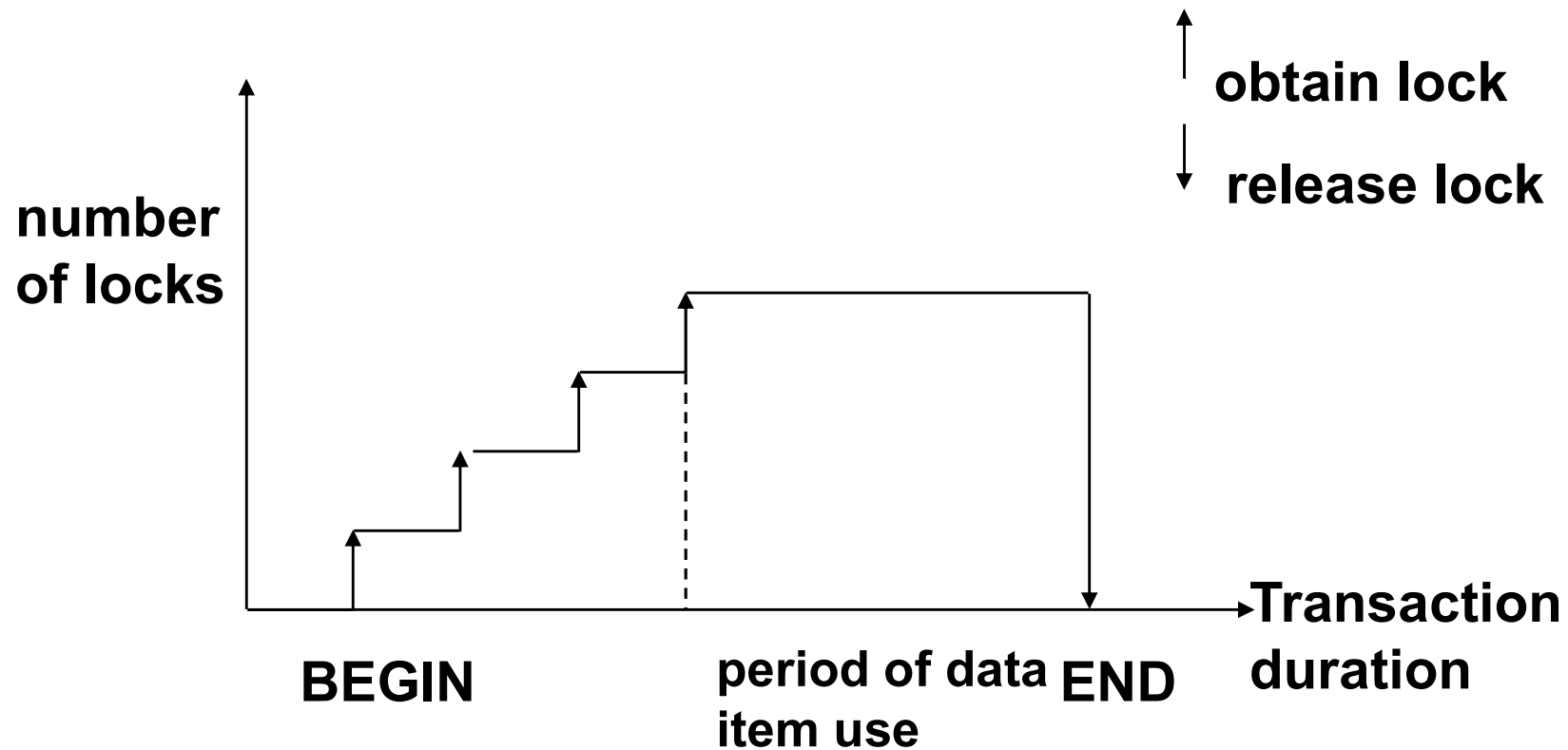


Conservative Two Phase Locking (C2PL)



- Conservative 2PL is a deadlock-free protocol. In Conservative 2PL, if a transaction T_i is waiting for a lock held by T_j , T_i is holding no locks (no hold and wait situation \Rightarrow no deadlock).
- However, it is difficult to use in practice because of the need to predeclare the read-set and write-set, which is not possible in some situations.

Strict Two Phase Locking (S2PL)



Comparisons

Protocol	Guarantee Serializability	Guarantee Recoverability	Prevent Deadlock	Concurrency
B2PL	YES	No	No	Higher
S2PL	YES	YES	No	Middle
C2PL	YES	YES	YES	Lower

Deadlock

- Deadlock occurs when each transaction T in a set of two or more transactions is waiting for some item that is locked by some other transaction T' in the set.
- Hence, each transaction in the set is in a waiting queue, waiting for one of the other transactions in the set to release the lock on an item.
- But because the other transaction is also waiting, it will never release the lock.

Deadlock Prevention

- Deadlock condition
 - ◆ Hold and wait
 - ◆ Cyclic wait
- Deadlock Prevention
 - ◆ A transaction locks all data items it refers to before it begins execution
 - ◆ The conservative two-phase locking uses this approach
 - ◆ This way of locking prevents deadlock since a transaction never hold and wait
 - ◆ But the concurrency is low

Deadlock Prevention using TS

- Suppose that transaction T_i tries to lock an item X but is not able to because X is locked by some other transaction T_j with a conflicting lock. The rules followed by these schemes are:
- **Wait-die** Rule : If $TS(T_i) < TS(T_j)$, then (T_i older than T_j) T_i is allowed to wait; otherwise (T_i younger than T_j) abort T_i (T_i dies) and restart it later with the same timestamp.
 - An older transaction is allowed to wait for a younger transaction, whereas a younger transaction requesting an item held by an older transaction is aborted and restarted.
- **Wound-Wait** Rule : If $TS(T_i) < TS(T_j)$, then (T_i older than T_j) abort T_j (T_i wounds T_j) and restart it later with the same timestamp; otherwise (T_i younger than T_j) T_i is allowed to wait
 - A younger transaction is allowed to wait for an older one, whereas an older transaction requesting an item held by a younger transaction preempts the younger transaction by aborting it.

Deadlock detection and resolution

■ Deadlock Detection

- ◆ In this approach, deadlocks are allowed to happen e.g., in Strict 2PL. The scheduler maintains a wait-for-graph for detecting cycle. If a cycle exists, then one transaction involved in the cycle is selected (victim) and rolled-back.
- ◆ One node is created in the wait-for graph for each transaction that is currently executing.
- ◆ Whenever a transaction T_i is waiting to lock an item X that is currently locked by a transaction T_j , a directed edge ($T_i \rightarrow T_j$) is created in the waitfor graph. When T_j releases the lock(s) on the items that T_i was waiting for, the directed edge is dropped from the wait-for graph.
- ◆ We have a state of deadlock if and only if the wait-for graph has a cycle.



■ End