
CS3402 Database Systems: ***Revision***

Exam info

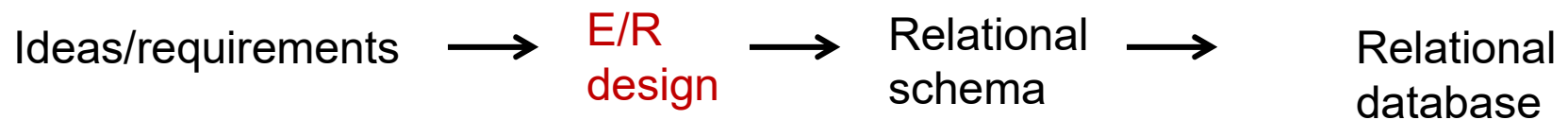
- Time:
- Duration: 2 hours
- Coverage: All lectures, all tutorials, all labs (focusing on the contents after midterm)
- Open book
- Allowed materials: Approved Electronic Calculator, Photocopy of book chapters, Printed lecture notes, Personal notes, Tutorial questions and answers, and other course handout materials

Exam info

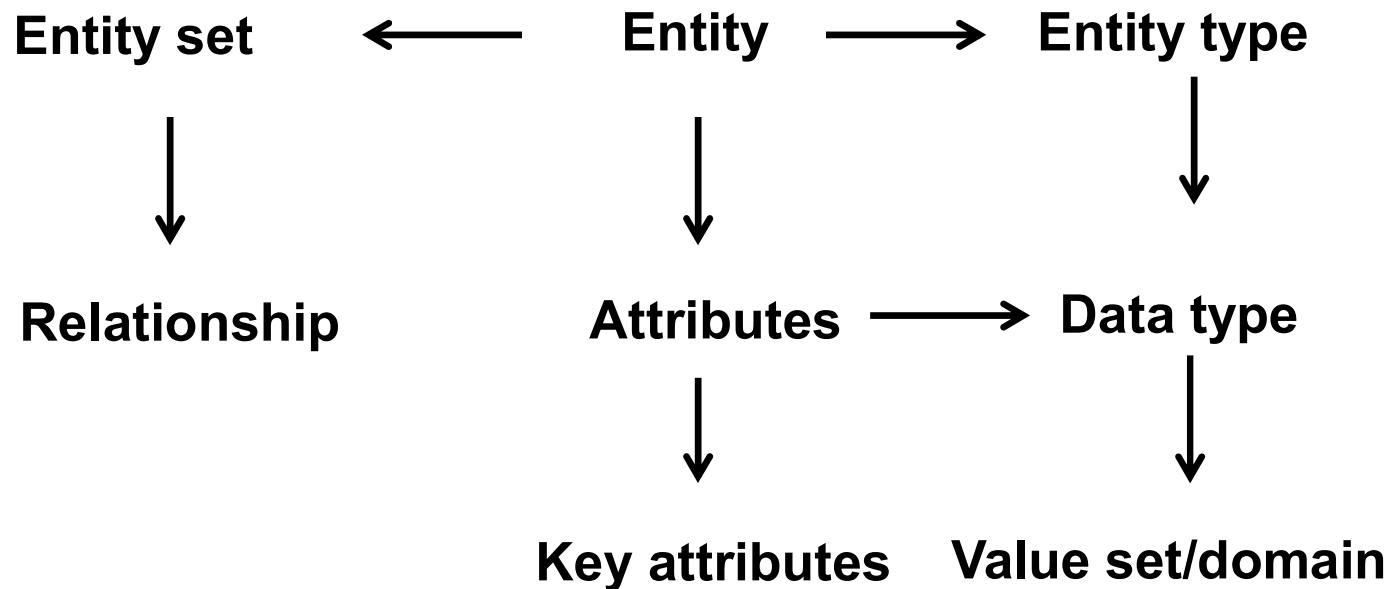
- Time: 8-May-19 09:30
- Duration: 2 hours
- Coverage: All lectures, all tutorials, all labs
- Open book
- Allowed materials: Approved Electronic Calculator, Photocopy of book chapters, Printed lecture notes, Personal notes, Tutorial questions and answers, and other course handout materials

Lec1: ER Model

Database Modelling and Implementation

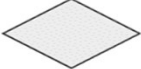
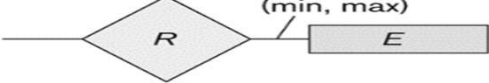


Basic Concepts



NOTATION for ER diagrams

Figure 3.14
Summary of the
notation for ER
diagrams.

Symbol	Meaning
	Entity
	Weak Entity
	Relationship
	Identifying Relationship
	Attribute
	Key Attribute
	Multivalued Attribute
	Composite Attribute
	Derived Attribute
	Total Participation of E_2 in R
	Cardinality Ratio 1: N for $E_1:E_2$ in R
	Structural Constraint (min, max) on Participation of E in R

Constraints on Relationships

- Constraints on Relationship Types

- ◆ Cardinality Ratio (specifies *maximum* participation)

- ◆ One-to-one (1:1), e.g., 1 employee manages 1 dept.

- ◆ One-to-many (1:N), e.g., 1 dept. controls N projects

- ◆ Many-to-one (N:1), e.g., N employees work for 1 dept.

- ◆ Many-to-many (M:N); e.g., M employees work on N projects

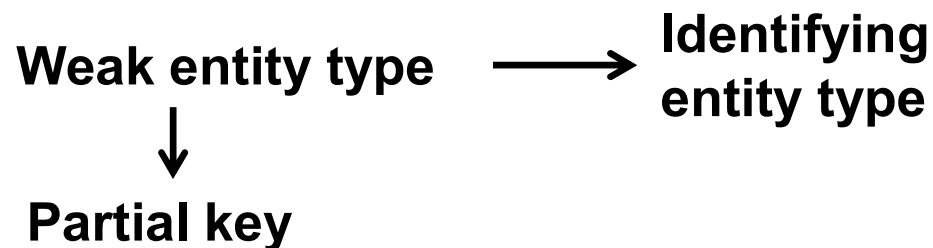
- ◆ Shown by placing appropriate numbers on the relationship edges

Participation Constraints on Relationships

- Participation constraint (on each participating entity type):
 - ◆ Specify the minimum no. of relationship instances that each entity can participate in
 - ◆ Total (existence dependency) or partial
 - ◆ Total shown by double line, partial by single line
 - ◆ E.g., double line: Dependent to Employee; single line: Employee to Dependent (not all employees have dependents)

Weak Entity Types

- An entity that does not have a key attribute and is identification-dependent on another entity type
- A weak entity must participate in an identifying relationship type with an owner or identifying entity type
- Entities are identified by the combination of:
 - ◆ A partial key of the weak entity type
 - ◆ The particular entity they are related to in the identifying relationship type

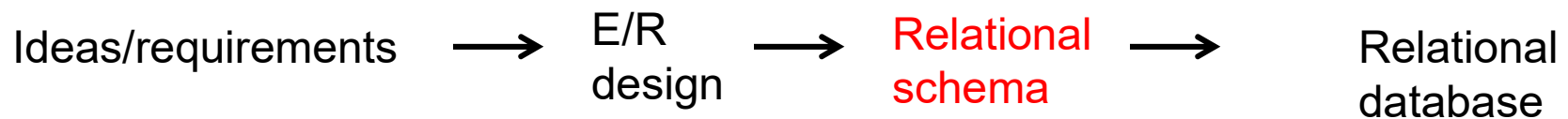


Summary

- ER concepts
- Entity, Attributes and Relationships
- Constraints on Relationships
- Multiplicity of Binary Relationship
- Weak Entity Type
- Alternative Notations
- Higher Degree of Relationships

Lec2: Relational Model

Database Modelling and Implementation



Informal Description

- A **relation** looks like a **table** (rows x columns) of values
- A relation contains a **set of rows (tuples)** and each **column (attribute)** has a column header that gives an indication of the **meaning** of the data items in that column
 - ◆ Associated with each attribute of a relation is a set of values (**domain**)
 - ◆ **Movies(title:string, year:integer, length:integer)**
- The data elements in each **row (tuple)** represent certain **facts** that correspond to a real-world **entity or relationship**

Formal Definition

<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

Populated Relation State

- Each *relation* has many records/tuples in its current relation state/instance
- Whenever the database is changed, a new state arises
- Basic operations for changing the database:
 - ◆ **INSERT** a new tuple in a relation
 - ◆ **DELETE** an existing tuple from a relation
 - ◆ **MODIFY** an attribute of an existing tuple

ER-to-Relational Mapping Algorithm

- COMPANY database example
 - ◆ Assume that the mapping will create **tables** with simple **single-valued attributes**
- Step 1: Mapping of Regular (strong) Entity Types
 - ◆ For each regular entity type, create a relation R that includes all the simple attributes of E
 - ◆ Choose one of the key attributes E as the primary key for R
 - ◆ Called **entity relations**
 - Each tuple represents an **entity instance**

ER-to-Relational Mapping Algorithm

- Step 2: Mapping of Weak Entity Types
 - ◆ For each weak entity type, create a relation R and include all simple attributes of the entity type as attributes of R
 - ◆ Include primary key attribute of owner as foreign key attributes of R
 - ◆ The primary key of R is the combination of the primary key of the owner and the partial key of the weak entity type
 - ◆ E.g., Essn and Dependent_Name

ER-to-Relational Mapping Algorithm

- Step 3: Mapping of Binary 1:1 Relationship Types
 - ◆ For each binary 1:1 relationship type
 - Identify relations that correspond to entity types participating in R
 - ◆ Possible approaches:
 - Foreign key approach
 - Merged relationship approach
 - Cross reference or relationship relation approach

ER-to-Relational Mapping Algorithm

- Foreign key approach (S – T)
 - ◆ Choose one of the relations S and include as a foreign key in S the primary key of T
 - ◆ Include all simple attributes as attributes of S
 - ◆ E.g., MANAGES (Mgr_ssn and Mgr_start_date) into DEPARTMENT

ER-to-Relational Mapping Algorithm

- Merged relationship approach
 - ◆ To merge the two entity types and the relationship into a single relation
- Cross reference or **relationship relation** approach
 - ◆ Set up a third relation R for the purpose of **cross-referencing** the primary keys of the two relations S and T representing the entity types
 - ◆ The relation R will include the **primary key attributes** of S and T as foreign keys to S and T
 - ◆ The primary key of R will be **one of the two foreign keys**

ER-to-Relational Mapping Algorithm

- Step 4: Mapping of Binary 1:N Relationship Types
 - ◆ For each regular binary 1:N relationship type
 - Identify relation that represents participating entity type at *N-side* of relationship type S
 - Include *primary key* of other entity type as *foreign key* in S
 - Include simple attributes of 1:N relationship type as attributes of S
 - E.g., for WORK_FOR, we include the primary key Dnumber of the DEPARTMENT as foreign key in EMPLOYEE
 - ◆ Alternative approach
 - Use the *relationship relation* (cross-reference) option as in the third option for binary 1:1 relationships

ER-to-Relational Mapping Algorithm

- Step 5: Mapping of Binary *M:N* Relationship Types
 - ◆ For each binary *M:N* relationship type
 - Create a new relation *S*
 - Include as **primary key** of participating entity types as foreign key attributes in *S*
 - Their combination forms the **primary keys** of the relations
 - Include any simple attributes of *M:N* relationship type
 - E.g., For WORKS_ON, we include the primary keys of PROJECT and EMPLOYEE as foreign keys in WORK_ON and rename them Pno and Essn, respectively

ER-to-Relational Mapping Algorithm

- Step 6: Mapping of Multivalued Attributes
 - ◆ For each multivalued attribute A
 - Create a new relation R
 - Primary key of R is the combination of A and K (the primary key attribute) of the relation that represents the entity type or relationship that has A as a multivalued attribute
 - If the multivalued attribute is composite, include its simple components
 - E.g., we create a relation DEPT_LOCATION. The attribute Dlocation represents the multivalued attributes LOCATIONS of DEPARTMENT
 - The primary key of DEPT_LOCATIONS is the combination of Dnumber and Dlocation

ER-to-Relational Mapping Algorithm

- Step 7: Mapping of N -ary Relationship Types
 - ◆ For each n -ary relationship type R
 - Create a new relation S to represent R
 - Include **primary keys** of participating entity types as foreign keys
 - Include any simple attributes as attributes
 - The primary key of S is a combination of **all foreign keys** that reference the relations representing the participating entity types

Example: ER of Company

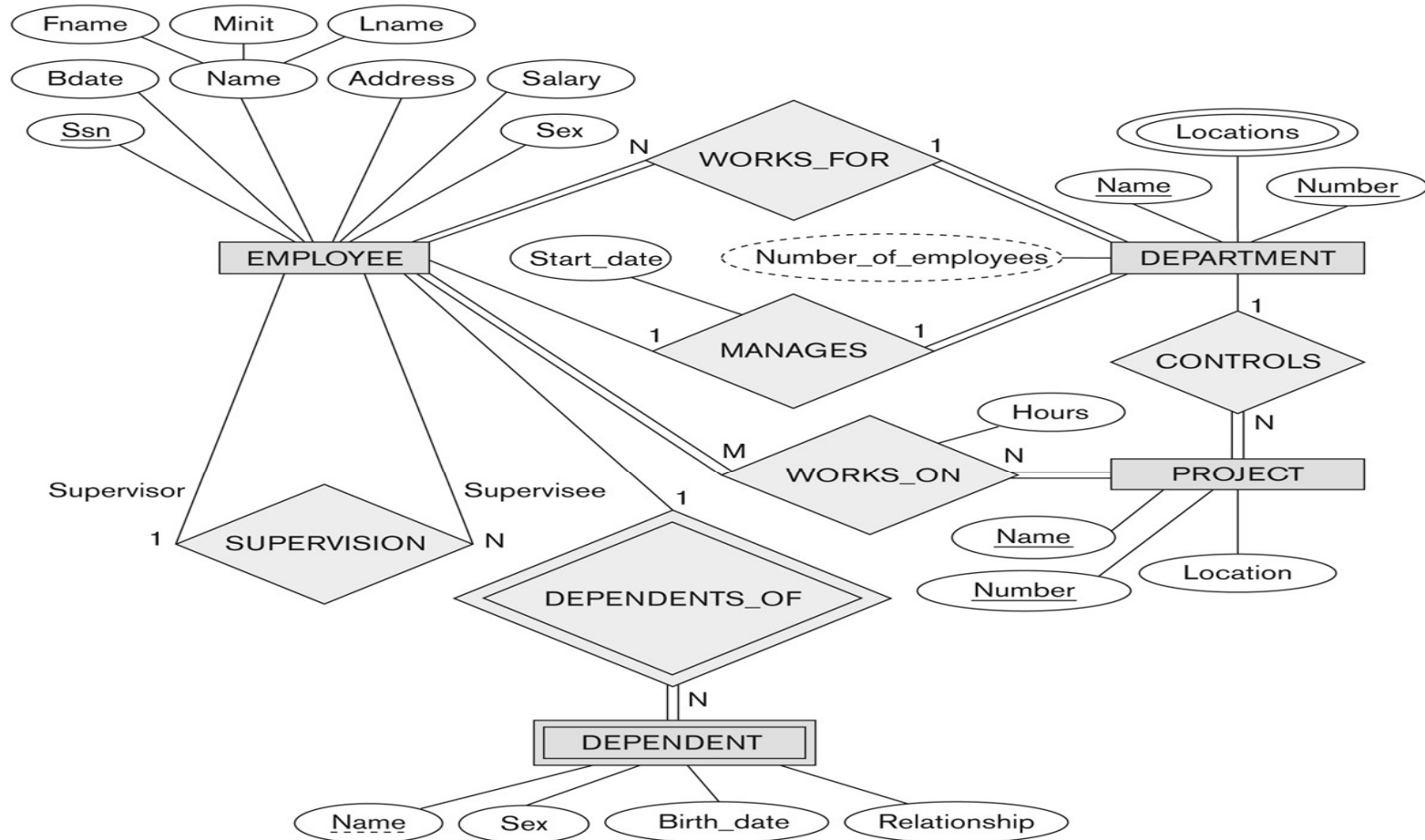


Figure 3.2

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>	<u>Plocation</u>	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.

Summary

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

SQL: Structured Query Language

- SQL is a hybrid language, its basically 4 types of languages in one
 - ◆ Data Definition Language(DDL)
used for define database schemas
 - ◆ Data Query Language (DQL)
used to query the database for information
 - ◆ Data Manipulation Language (DML)
used for inserting, updating and deleting data from the database
 - ◆ Data Control Language (DCL)
used for controlling the access to the data in the database
- Each statement in SQL ends with a semicolon (;)
- SQL is case insensitive
- Implementations of SQL in Different DBMS (MySQL, Oracle) are slightly different

DDL:CREATE Table

- Within a schema, we have tables
- Create TABLE: optionally specify schema
 - ◆ CREATE TABLE COMPANY.EMPLOYEE ... or
 - ◆ CREATE TABLE EMPLOYEE ...
- Base tables (base relations)
 - ◆ 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 relations
 - ◆ Created through the CREATE VIEW statement
 - ◆ The tables are **not actually created** but are presented to the user through reconstruction (view) of base tables

DDL:Table Manipulation

- Table creation: **CREATE TABLE** table-name
(A1, D1,
A2, D2, ...
Ak, Dk);

- Table deletion: **DROP TABLE** table-name;

- Table update:
ALTER TABLE table-name
ADD Aj, Dj

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

DQL: The *SELECT-FROM-WHERE* Structure of Basic SQL Queries

■ *SELECT* statement

◆ The basic statement for retrieving information from a database

SELECT <attribute list>
FROM <table list>
WHERE <condition>;

where

SELECT <attribute list>
FROM <table list>
[**WHERE** <condition>]
[**ORDER BY** <attribute list>];

- <attribute list> is a list of attribute names whose values are to be retrieved by the query.
- <table list> is a list of the relation names required to process the query.
- <condition> is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query.

Specified attributes



Satisfy the
conditions



Title	Year	Length	Type
Star War	1977	124	Color
Mighty Duck	1991	104	Color
Wayne's World	1992	95	Color

Aliasing, and Renaming

- Aliases or tuple variables

- ◆ Declare alternative relation names E and S to refer to the EMPLOYEE relation twice in a query:

Query 8. For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.

```
SELECT E.Fname, E.Lname, S.Fname, S.Lname
FROM EMPLOYEE AS E, EMPLOYEE AS S
WHERE E.Super_ssn=S.Ssn;
```

One level
recursive query

Alias

- ◆ Recommended practice to abbreviate names and to prefix same or similar attribute from multiple tables

Unspecified WHERE Clause and Use of the Asterisk

- Specify an asterisk (*)
 - ◆ Retrieve all the attribute values of the selected tuples

Q1C: SELECT *
 FROM EMPLOYEE
 WHERE Dno=5;

Q1D: SELECT *
 FROM EMPLOYEE, DEPARTMENT
 WHERE Dname='Research' AND Dno=Dnumber;

Q10A: SELECT *
 FROM EMPLOYEE, DEPARTMENT;

←
EMPLOYEE: 100 tuples
DEPARTMENT: 10 tuples
Finally, 1000 tuples and all attributes

DISTINCT in the SELECT clause

- SQL does not automatically eliminate duplicate tuples (the attributes of two tuples have same values) in query results (**NOT** a set)
- Use the keyword **DISTINCT** in the **SELECT** clause
 - ◆ Only distinct tuples should remain in the result

Query 11. Retrieve the salary of every employee (Q11) and all distinct salary values (Q11A).

Q11: **SELECT** **ALL** Salary
 FROM **EMPLOYEE;**

Q11A: **SELECT** **DISTINCT** Salary
 FROM **EMPLOYEE;**

Ordering Tuples

- List in alphabetic order all customers having a loan at Kowloon branch

```
SELECT DISTINCT cname  
FROM Borrow  
WHERE bname = "Kowloon"  
ORDER BY cname;
```

By default, in ascending order.

- List the entire borrow table in descending order of amount, and if several loans have the same amount, order them in ascending order by loan#

```
SELECT *  
FROM Borrow  
ORDER BY amount DESC, loan# ASC;
```

Nested Queries

Q4A: SELECT DISTINCT Pnumber
 FROM PROJECT
 WHERE Pnumber IN
 (SELECT Pnumber
 FROM PROJECT, DEPARTMENT, EMPLOYEE
 WHERE Dnum=Dnumber AND
 Mgr_ssn=Ssn AND Lname='Smith')

 OR
 Pnumber IN
 (SELECT Pno
 FROM WORKS_ON, EMPLOYEE
 WHERE Essn=Ssn AND Lname='Smith');

The project no. of
projects that have an
employee with last
name "Smith"



Tables as Sets in SQL

■ Set operations

◆ UNION, EXCEPT (difference), INTERSECT

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

EXPANDED Block 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> ];
```

GROUP BY and HAVING Clauses

- **HAVING clause**

- ◆ Provides a condition to select or reject an entire group:

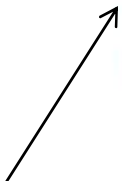
- **Query 26.** For each project on which more than two employees work, retrieve the project number, the project name, and the number of employees who work on the project.

Q26:	SELECT	Pnumber, Pname, COUNT (*)
	FROM	PROJECT, WORKS_ON
	WHERE	Pnumber=Pno
	GROUP BY	Pnumber, Pname
	HAVING	COUNT (*) > 2;

GROUP BY and HAVING Clauses

Query 28. For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than \$40,000.

Q28: **SELECT** Dnumber, COUNT (*)
 FROM DEPARTMENT, EMPLOYEE
 WHERE Dnumber=Dno **AND** Salary>40000 **AND**
 (**SELECT** Dno
 FROM EMPLOYEE
 GROUP BY Dno
 HAVING COUNT (*) > 5)



The employees with the same Dno are group and then count the number of employees in each group. If the number is greater than 5, put the value into the SELECTED tuples

GROUP BY

- Some Example Queries:

- ◆ Find the average account balance at each branch

```
SELECT b-name, AVG(balance)
```

```
FROM Deposit
```

```
GROUP BY b-name;
```

← AVG(balance) is the
average of each branch

- ◆ If only interested in branches where average balance
is > \$12000

```
SELECT b-name, AVG(balance)
```

```
FROM Deposit
```

```
GROUP BY b-name
```

```
HAVING AVG(balance) > 12000;
```

Aggregate Functions

- Used to summarize information from multiple tuples into a single-tuple
- Grouping
 - ◆ Create subgroups of tuples before summarizing
 - ◆ E.g., A group of tuples => Count its number to be the value of a new attribute
- Built-in aggregate functions
 - ◆ COUNT, SUM, MAX, MIN, and AVG
- Functions can be used in the SELECT clause or in a HAVING clause
- If NULLs exist in grouping attribute
 - ◆ Separate group created for all tuples with a NULL value in grouping attribute

DML: INSERT Command

- 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

DML: DELETE Command

- Removes tuples from a relation
 - ◆ Includes a `WHERE` clause to select the tuples to be deleted

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;

Delete all tuples but not the table

DML: UPDATE Command

- 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;
```

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.

Lec4: 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))
 - ◆ Relational algebra Operations from Set theory
 - ◆ UNION ($\cup$), INTERSECTION ($\cap$), DIFFERENCE (or MINUS, $-$)
 - ◆ CARTESIAN PRODUCT ($\times$)
 - ◆ Binary Relational Operations
 - ◆ JOIN (several variations of JOIN exist)
 - ◆ DIVISION
 - ◆ 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*)

```
SELECT *  
FROM EMPLOYEE  
WHERE DNO=4
```

- Examples:
 - ◆ Select the EMPLOYEE tuples whose department number is 4:

$\sigma_{DNO=4}(EMPLOYEE)$

- ◆ Select the employee tuples whose salary is greater than \$30,000.

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
- Example: To list each employee's first and last name and salary, the following is used:

$\pi_{\text{LNAME, FNAME, SALARY}}(\text{EMPLOYEE})$

SELECT LNAME, FNAME, SALARY
FROM EMPLOYEE

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 $B_1, B_2, \dots, B_n$
 - ◆ $\rho_S(R)$ changes:
 - ◆ the *relation name* only to S
 - ◆ $\rho_{(B_1, B_2, \dots, B_n)}(R)$ changes:
 - ◆ the *attribute names* only to $B_1, B_2, \dots, B_n$

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

Set Theory: UNION

- UNION Operation

- ◆ Binary operation, 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

- ◆ Duplicate tuples are eliminated

- ◆ The two operand relations R and S must be “type compatible” (or UNION compatible)

- ◆ R and S must have same number of attributes

- ◆ Each pair of corresponding attributes must be type compatible (have same or compatible domains)

Relational Algebra Operations from Set Theory: INTERSECTION

- INTERSECTION is 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
 - ◆ The attribute names in the result will be the same as the attribute names in R
- The two operand relations R and S must be “type compatible”

Relational Algebra Operations from Set Theory: SET DIFFERENCE

- SET DIFFERENCE (also called MINUS or EXCEPT) is denoted by –
- The result of $R - S$, is a relation that includes all tuples that are in R but not in S
 - ◆ The attribute names in the result will be the same as the attribute names in R
- The two operand relations R and S must be “type compatible”
- $R \cap S = (R \cup S) - (R - S) - (S - R)$

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 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*

Binary Relational Operations: Theta JOIN

- The general case of JOIN operation is called a **Theta-join**:
- $R \bowtie S$
- The join condition is called *theta*
- *Theta* can be any general **boolean expression** on the attributes of R and S; for example:
 - ◆ $R.A_i < S.B_j \text{ AND } (R.A_k = S.B_l \text{ OR } R.A_p < S.B_q)$
- Most join conditions involve one or more equality conditions “AND”ed together; for example:
 - ◆ $R.A_i = S.B_j \text{ AND } R.A_k = S.B_l \text{ AND } R.A_p = S.B_q$

Binary Relational Operations: EQUIJOIN

- EQUIJOIN Operation
- The most common use of join involves join conditions with *equality comparisons only*
- Such a join, where the only comparison operator used is $=$, is called an EQUIJOIN
 - ◆ In the result of an EQUIJOIN we always have one or more pairs of attributes (whose names need not be identical) that have identical values in every tuple
 - ◆ The JOIN seen in the previous example was an EQUIJOIN

Binary Relational Operations: 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
- ◆ If this is not the case, a renaming operation is applied first

Binary Relational Operations: DIVISION

■ DIVISION Operation

- ◆ The division operation is applied to two relations
- ◆ $R(Z) \div S(X)$, where X subset 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

Summary: Operations of Relational Algebra

Table 8.1 Operations of Relational Algebra

OPERATION	PURPOSE	NOTATION
SELECT	Selects all tuples that satisfy the selection condition from a relation R .	$\sigma_{\langle \text{selection condition} \rangle}(R)$
PROJECT	Produces a new relation with only some of the attributes of R , and removes duplicate tuples.	$\pi_{\langle \text{attribute list} \rangle}(R)$
THETA JOIN	Produces all combinations of tuples from R_1 and R_2 that satisfy the join condition.	$R_1 \bowtie_{\langle \text{join condition} \rangle} R_2$
EQUIJOIN	Produces all the combinations of tuples from R_1 and R_2 that satisfy a join condition with only equality comparisons.	$R_1 \bowtie_{\langle \text{join condition} \rangle} R_2$, OR $R_1 \bowtie_{(\langle \text{join attributes 1} \rangle), (\langle \text{join attributes 2} \rangle)} R_2$
NATURAL JOIN	Same as EQUIJOIN except that the join attributes of R_2 are not included in the resulting relation; if the join attributes have the same names, they do not have to be specified at all.	$R_1^*_{\langle \text{join condition} \rangle} R_2$, OR $R_1^*_{(\langle \text{join attributes 1} \rangle), (\langle \text{join attributes 2} \rangle)} R_2$ OR $R_1 * R_2$

Summary: Operations of Relational Algebra

Table 8.1 Operations of Relational Algebra

OPERATION	PURPOSE	NOTATION
UNION	Produces a relation that includes all the tuples in R_1 or R_2 or both R_1 and R_2 ; R_1 and R_2 must be union compatible.	$R_1 \cup R_2$
INTERSECTION	Produces a relation that includes all the tuples in both R_1 and R_2 ; R_1 and R_2 must be union compatible.	$R_1 \cap R_2$
DIFFERENCE	Produces a relation that includes all the tuples in R_1 that are not in R_2 ; R_1 and R_2 must be union compatible.	$R_1 - R_2$
CARTESIAN PRODUCT	Produces a relation that has the attributes of R_1 and R_2 and includes as tuples all possible combinations of tuples from R_1 and R_2 .	$R_1 \times R_2$
DIVISION	Produces a relation $R(X)$ that includes all tuples $t[X]$ in $R_1(Z)$ that appear in R_1 in combination with every tuple from $R_2(Y)$, where $Z = X \cup Y$.	$R_1(Z) \div R_2(Y)$

Lec 5: Integrity Constraints

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

Relational Integrity Constraints

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

Key Constraints

- Superkey of R:

- ◆ A set of attributes that **contains a key** is called a **superkey**
- ◆ It is a set of attributes SK, e.g., {A1, A2} of R with the following condition:
 - ◆ 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 $r(R)$, $t1[SK] \neq t2[SK]$

- Key (Primary key, 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)

Key Constraints

- If a relation has several **candidate keys**, one is chosen arbitrarily to be the **primary key**
 - ◆ The primary key attributes are underlined
- Example: Consider the CAR relation schema:
 - ◆ CAR(State, Reg#, SerialNo, Make, Model, Year)
 - ◆ We chose SerialNo as the primary key
- The primary key value is used to *uniquely identify* each tuple in a relation
- General rule: Choose as primary key the smallest of the candidate keys (in terms of size)

Entity Integrity Constraints

- Entity Integrity:

- ◆ The *primary key attributes* PK of each relation schema R cannot have null values in any tuple of R
 - ◆ Primary key values are used to *identify* the individual tuples
 - ◆ $t[PK] \neq \text{null}$ for any tuple t in R
 - ◆ If PK has several attributes, null is not allowed in any of these attributes
- ◆ Note: Other attributes of R may be constrained to disallow null values, even though they are not members of the primary key

Domain Constraints

- ◆ *Domain constraint*: Every value in a tuple must be from the *domain of its attribute* (or it could be **null**, if allowed for that attribute)

E.g.,

C-Name: string of char (30)

Balance: Number (6,2)

...

Referential Integrity

- Key, domain, entity integrity constraints are specified on individual relations
- Referential integrity is a constraint involving two relations
 - ◆ To specify a relationship among tuples in two relations
 - ◆ The referencing relation and the referenced relation ($R1 \rightarrow R2$)
- Tuples in the referencing relation $R1$ have attributes FK (called foreign key attributes) that reference the primary key attributes PK of the referenced relation $R2$ if it satisfies:
 - ◆ The attributes in FK have the same domain(s) as the primary key attributes PK of $R2$

Displaying a relational database schema and its constraints

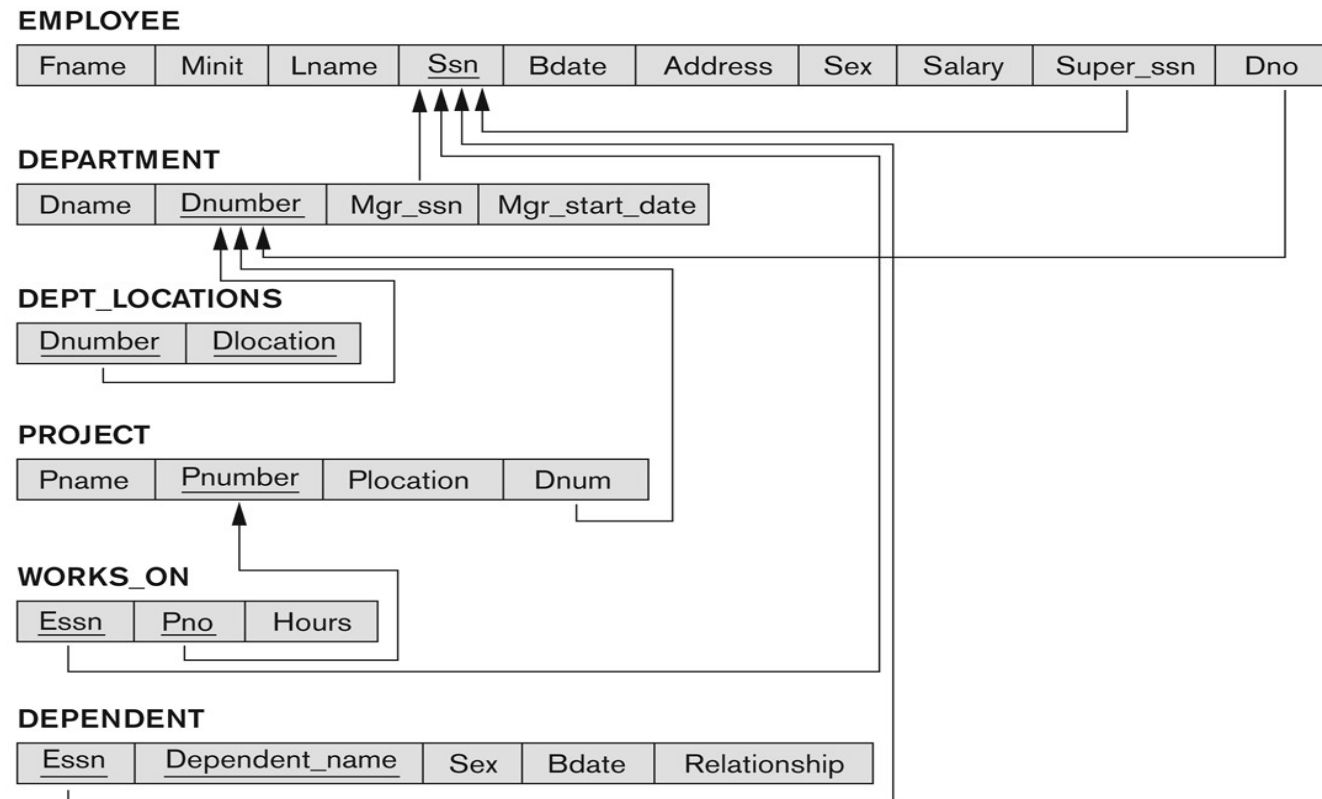
- Each relation schema can be displayed as a row of attribute names
- The name of the relation is written above the attribute names
- The primary key attribute (or attributes) will be underlined
- A foreign key (referential integrity) constraints is displayed as a directed arc (arrow) from the foreign key attributes to the referenced table
- Next slide shows the COMPANY relational schema diagram with referential integrity constraints

Database State for COMPANY

- All examples discussed below refer to the COMPANY database shown here

Figure 5.7

Referential integrity constraints displayed on the COMPANY relational database schema.



Functional Dependency

- Formal definition:

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

$$\alpha \rightarrow \beta$$

- ◆ If in any relation instance $r(R)$, for all pairs of tuples $t1$ and $t2$ in r , we have:

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

- E.g., for the table Borrow (B-name, Loan#, C-name, Amount), we have: $\text{Loan\#} \rightarrow \text{Amount}$
(read as “uniquely determines”, “Functional determines”)

Inference Rules for FDs

IR1 (reflective rule)	If X is a subset of Y , 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 and equivalence of FDs

- Closure of a set F of FDs
 - ◆ The set F^+ of all FDs that can be inferred from F
- Closure of a set of attributes X with respect to F
 - ◆ The set X^+ of all attributes that are functionally determined by X
- A set of functional dependencies F is said to cover another set of functional dependency E
 - ◆ If every FD in E is also in F^+
- Two sets of FDs F and G are equivalent
 - ◆ If F and G are equivalent if $F^+ = G^+$

Lec 7: Nomalization

- Normalization
 - ◆ Proposed by Codd (1972): take a relation schema through a series of tests to certify whether it satisfies a certain **normal form**
- Analyzing the relation scheme based on **FD** and **primary keys** to achieve:
 - Minimizing redundancy
 - Minimizing the insertion, deletion and update anomalies

Normalization

- Normalization requires two properties

- ◆ Non-additive or lossless join

- Decomposition is **reversible** and no information is lost
 - No **spurious tuples** (tuples that should not exist) should be generated by doing a natural-join of any relations
(Extremely important)

- ◆ Preservation of the functional dependencies

- Ensure each functional dependency is represented in some individual relation (Sometimes can be sacrificed)

First Normal Form with Primary Key

- First normal form (1NF)
 - ◆ Disallow multi-values attributes, composite attributes and their combination
 - ◆ The domain of an attribute **must be atomic** (simple and indivisible) values
- DEPARTMENT: each department can have a number of locations
- 1NF: DEPT_LOCATIONS(Dnumber, Dlocation)
- 1NF also disallows multivalued attributes that are themselves composite

Second Normal Form with Primary Key

- A relation schema R is in 2NF if every **nonprime attributes** A in R is **fully functional dependent** on the **primary key** of R
- An attribute of R is called **prime attribute** of R if it is a member of some **candidate key** of R. Otherwise it is nonprime
- Non-prime attributes: Hours, Ename, Pname, Plocation
- Primary key: {Ssn, Pnumber}

Third Normal Form with Primary Key

- According to Codd's original definition, a relation scheme R is in 3NF if it satisfies 2NF and no **non-prime attribute of R** is **transitively dependent** on the **primary key**
- 3NF: ED1(Ename, Ssn, Bdate, Address, Dnumber); ED2(Dnumber, Dname, Dmgr_ssn)



Normal Forms Defined Informally

- 1st normal form
 - ◆ All attributes depend on the key (no nested relations or multivalued attributes)
- 2nd normal form
 - ◆ All non-key attributes depend on the whole key (no partial dependency)
- 3rd normal form
 - ◆ All non-key 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 Normal Forms

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).

Boyce-Codd Normal Form

- BCNF was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF
- Every relation in BCNF is also in 3NF
 - ◆ BUT Relation in 3NF is not necessarily in BCNF

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 .

- Difference:
 - ◆ Condition which allows A to be prime is absent from BCNF

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 .

Lec 8 : File

Records

Blocks

Files

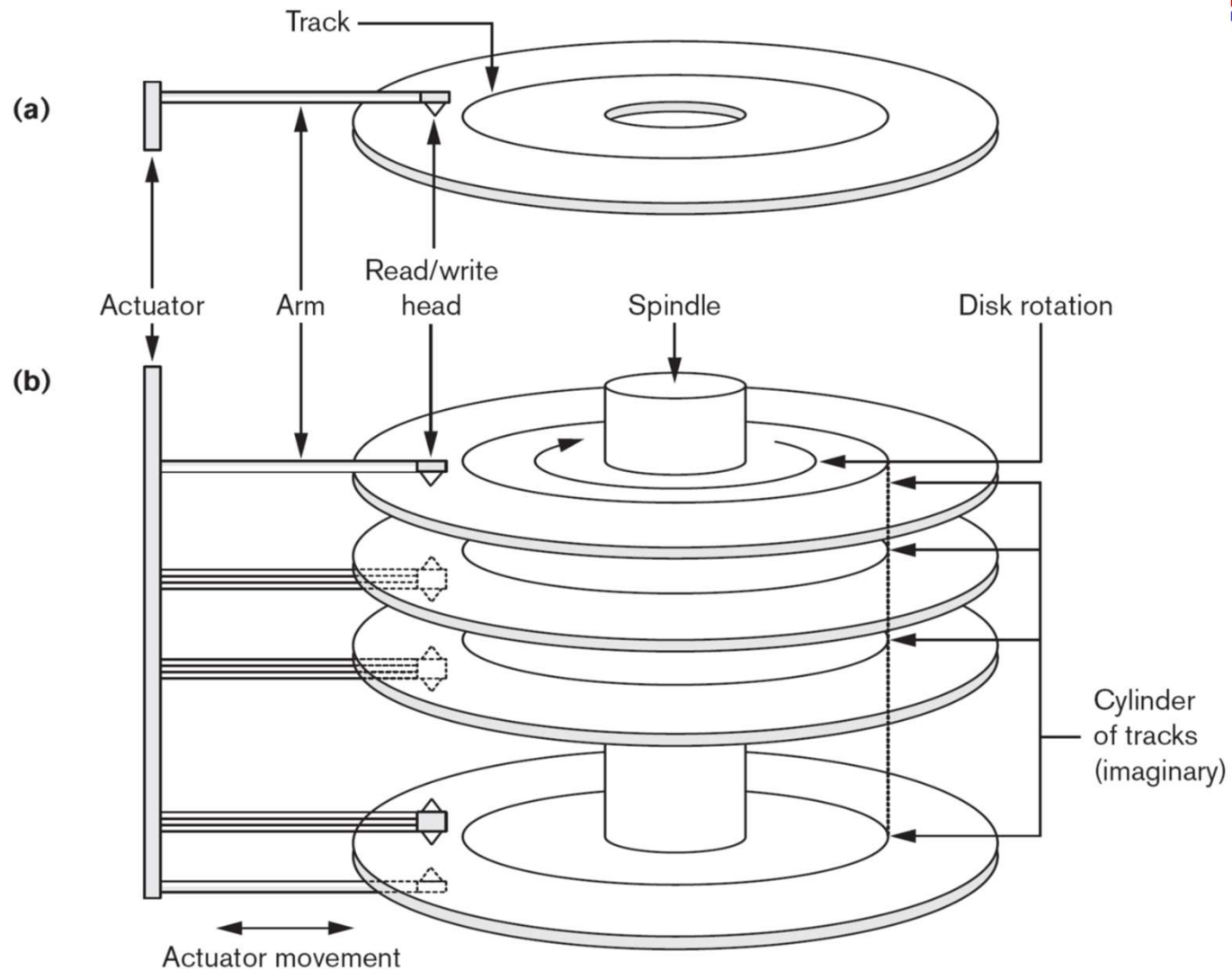
Storage Medium for Databases

- Memory hierarchy
 - ◆ CPU cache => main memory => flash memory/Phase change memory => magnetic disks/optical disks
 - ◆ Slower in access delay but larger in memory size (less expensive)
 - ◆ Volatile vs. non-volatile
- Primary storage
 - ◆ The storage media that can be operated directly by the CPU
 - ◆ Include main memory and cache memory
- Secondary and tertiary storage
 - ◆ Slower in access
 - ◆ Include magnetic disks, optical disks and flash memory
- A database could be huge in size (several GB or even larger)
 - ◆ Need to be resided in secondary/tertiary memory (**persistent data and non-volatile storage**)

Disk Storage Devices

Figure 17.1

- (a) A single-sided disk with read/write hardware.
- (b) A disk pack with read/write hardware.



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
- A **linear search** through the file records is necessary to search for a record
 - ◆ This requires reading and searching half the file blocks on **average**, and is hence quite expensive ($n/2$)
 - ◆ Worst case, all records (n)
- Record insertion is efficient (add to the end)
- Reading the records in order of a particular field requires sorting the file records

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**
- **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 periodically merged with the main ordered file
- A **binary search** can be used to search for a record on its **ordering field** value
 - ◆ This requires reading and searching $\log_2 n$ of the file blocks on the average, an improvement over linear search
- Reading the records in **order of the ordering field** is quite **efficient**

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

Hashed Files

- Hashing for disk files is called External Hashing (files on disk)
- The file blocks are divided into M equal-sized buckets, numbered $\text{bucket}_0, \text{bucket}_1, \dots, \text{bucket}_{M-1}$
- One of the file fields is designated to be the hash key of the file
- Suppose there is a hash function that takes a hash key as an argument to compute an integer in the range 0 to $B - 1$ where B is the number of buckets
- A bucket array (an array index) from 0 to $B - 1$ holds the headers of B lists, one for each bucket of the array
- If a record has search key K , we store the record by linking it to the bucket list for the bucket numbered $h(K)$ where h is the hash function
 - ◆ $h(K) \Rightarrow 0 \text{ to } B - 1$

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 Bucket_id+1, Bucket_id+2, ..., Bucket_id+n
 - ◆ Quadratic Probing
 - ◆ If collide, try Bucket_id+1, Bucket_id+4, ..., Bucket_id+n²
 - ◆ 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

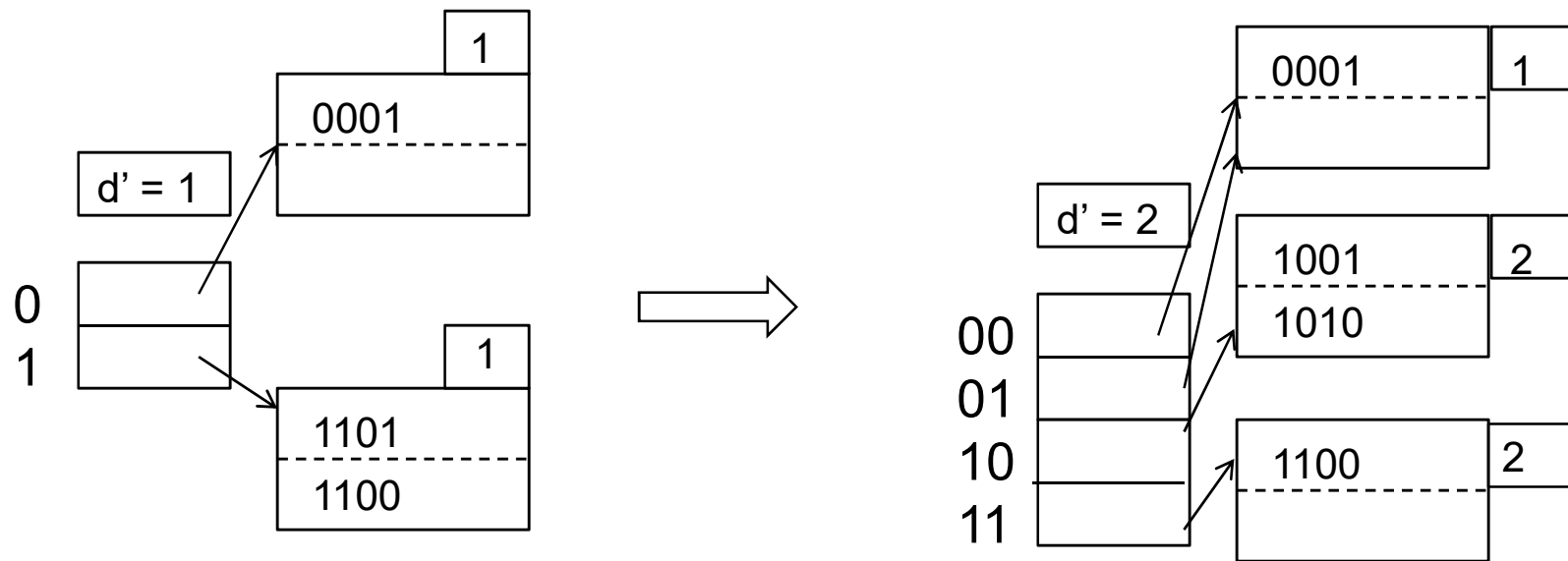
- Dynamic and Extendible Hashing Techniques
 - ◆ Hashing techniques are extended to allow dynamic growth and shrinking of the number of file records
 - ◆ These techniques include the following: dynamic hashing and extendible hashing
- Both dynamic and extendible hashing use the binary representation (e.g., 1100...) of the hash value $h(K)$ in order to access a directory
 - ◆ In dynamic hashing the directory is a binary tree
 - ◆ In extendible hashing the directory is an array of size 2^d where d is called the global depth

Extendible Hashing Example

- Suppose $d = 4$, i.e., the hash function produces a sequence of four bits
- At the moment, only one of these bits is used as illustrated by $i = 1$ ($d' = 1$) in the box above the bucket array
- The bucket array therefore has only two entries and points to two blocks
 - ◆ The first holds all the current records whose search keys hash to a bit sequence **beginning with 0**
 - ◆ The second holds all those whose search keys hash to a sequence **beginning with 1**
- Suppose we insert a record whose key hash to the sequence 1010
- Since the first bit is 1, it belongs in the second block
- However, the second block is full. It needs to be split by setting d' to 2

Extendible Hashing Example

- The two entries beginning with 0 each point to the block for records whose hashed keys begin with 0 and the block still has the integer 1 in its “nub” to indicate that only the first bit determine membership in the block
- The blocks for records beginning with 1 needs to be split into 10 and 11



Dynamic Hashing

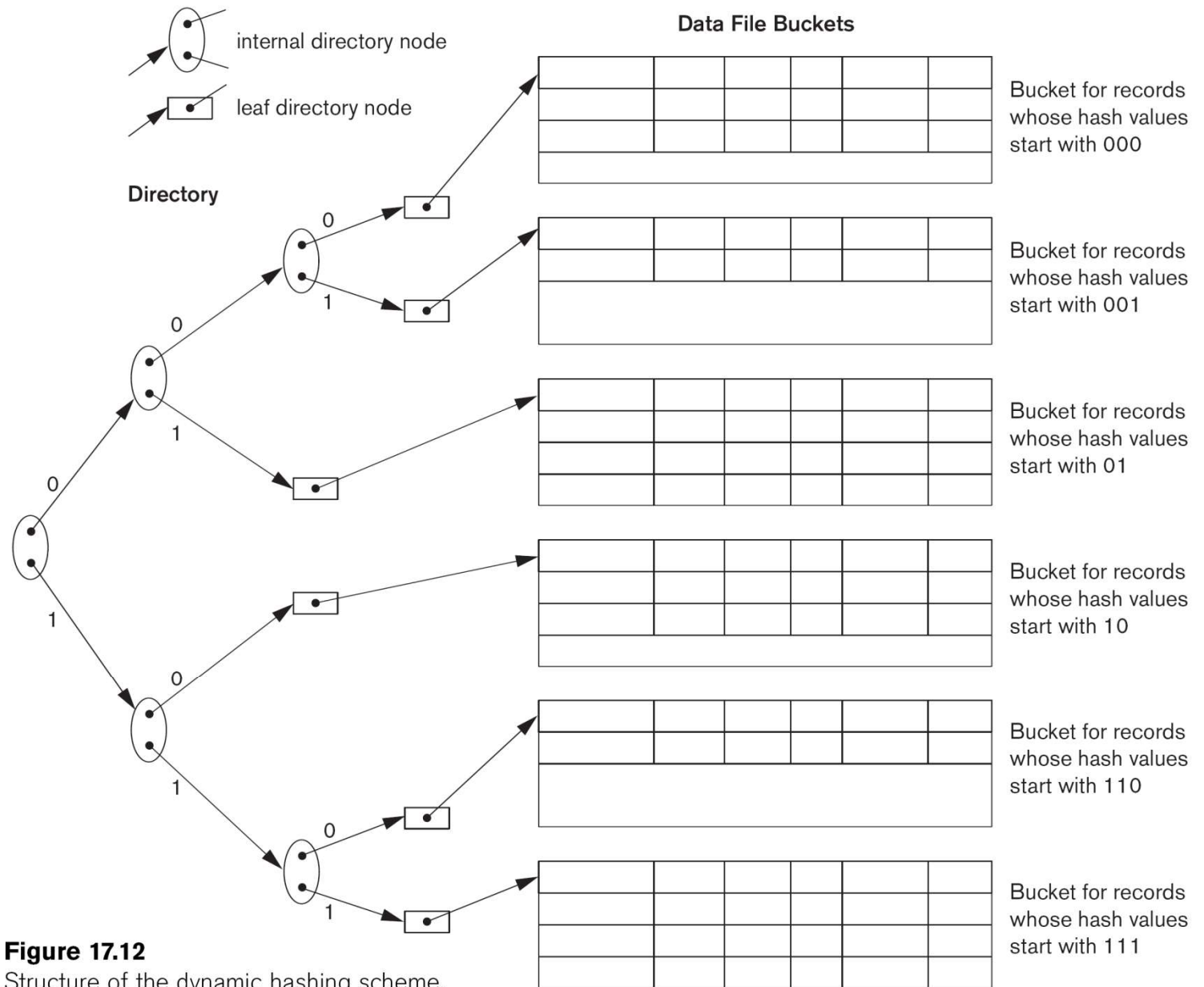


Figure 17.12
Structure of the dynamic hashing scheme.

Lec 9: Indexing

- An index is an **auxiliary file** to provides fast access to a record in a data file (index file + data file)
- Without an index, we may use the **sequential search** (for unordered file), i.e., to examine the record one by one to find the required record stored in disk (from beginning to end of the file until the required record is found)
- An index is an **ordered** sequence of entries <field value, pointer to record>, **ordered** by the field value
- The pointer provides the **address** (location or **physical block**) of the required record (in the data file) in disk storage
- An index is usually specified on one field (called **key field** which **may not be a key**) of the data file (although it could be specified on several fields)

Indexes as Access Paths

- The index file usually occupies considerably less disk blocks than the data file because its entries are much smaller
 - ◆ The biggest delay in data retrieval is the disk retrieval time
 - ◆ The size of each index entry is much smaller than the size of a record
 - ◆ Reading the index from disk is faster than reading data/records file from disk (smaller number of disk blocks)
 - ◆ Searching data/records in the main memory is very fast << the disk access
- Instead of searching the data file sequentially, we search the index to get the address of the required records
- A binary search on the index yields (ordered) a pointer to the file record

Types of Single Level Indexes

- Single level index: one index points to one data record
- Primary index (key field)
 - ◆ Specified on the ordering key field of an ordered file of records
- Cluster index (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 search key value (and hence every record) in the data file. Thus larger index size
 - ◆ A **sparse index**, on the other hand, has index entries for only some of the search values. Thus smaller index size

Properties of Index Types

Table 18.1 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 18.2 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 or Nondense	No

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 first-level index (primary, secondary, clustering) as long as the first-level index consists of **more than one** disk block

Tree Index Structure

- A multi-level index is a form of *search tree*
 - ◆ However, *insertion and deletion* of new index entries is a severe performance problem because every level of the index is an *ordered file*
- We may use a *B-/B⁺-tree index* to resolve the problem (reservation of nodes)
- A *tree* is formed of multi-level *nodes*
- Except the *root node*, each node in the tree has one *parent node* and zero or more *child node*
- A node that does not have any child nodes is called a *leaf node*
- A non-leaf node is called *internal node*
- A sub-tree of a node consists of the node and all its descendant
- If the leaf nodes are at different levels, the tree is called *unbalanced*

Dynamic Multilevel Indexes Using B-Trees and B+-Trees

- Most multi-level indexes use B-tree or B⁺-tree data structures for solving the insertion and deletion problem
 - ◆ They reserves spaces in each tree node (disk block) to allow for new index entries
- In B-Tree and B⁺-Tree, each node corresponds to a disk block
- Each node is kept between half-full and completely full
 - ◆ Need to restructure the tree in case of node overflow (insertion when full) or underflow (deletion when half full)
 - ◆ Less than half-full, wastes of space and the tree will have many levels (higher retrieval cost)

Difference between B-tree and B+-tree

- In a B-tree, pointers to data records exist at **all levels** of the tree
- In a B⁺-tree, all pointers to data records exists at the **leaf-level** nodes
- A B⁺-tree can have **less levels** (or **higher capacity of search values**) than the corresponding B-tree since its entry is **smaller in size**
- The internal nodes of B⁺-tree contains pointers only

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**

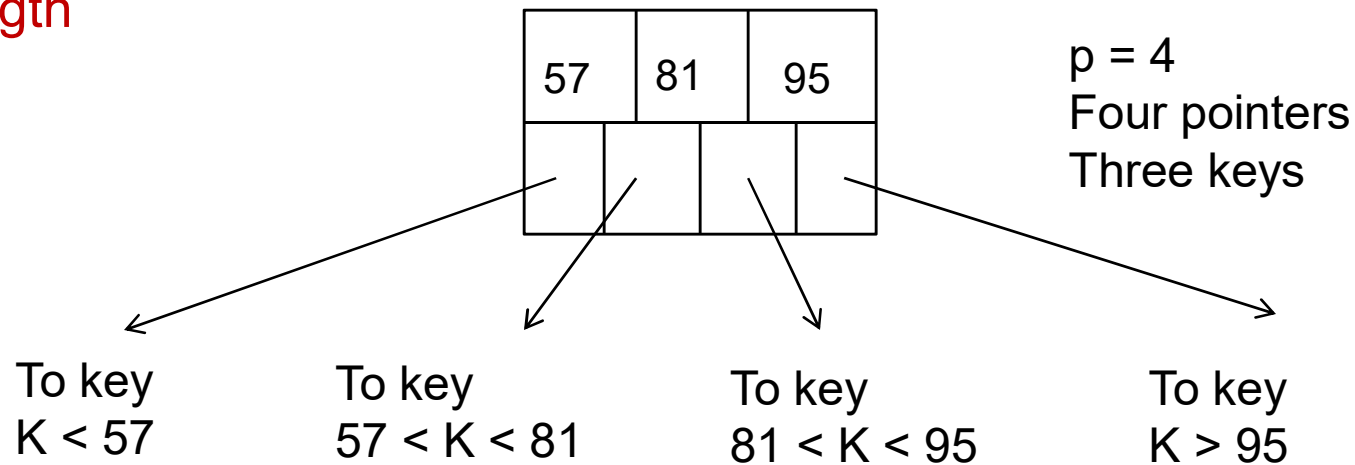
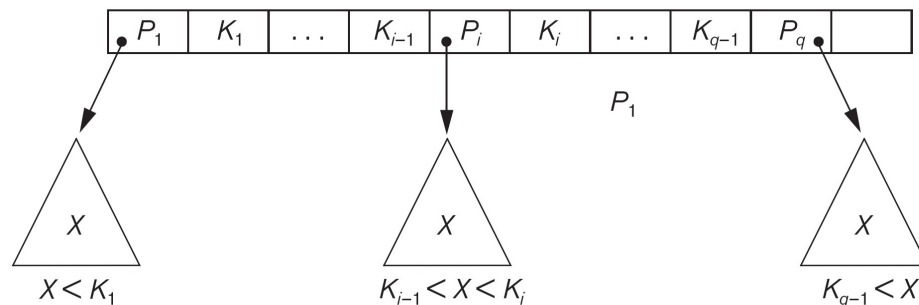


Figure 18.8

A node in a search tree with pointers to subtrees below it.



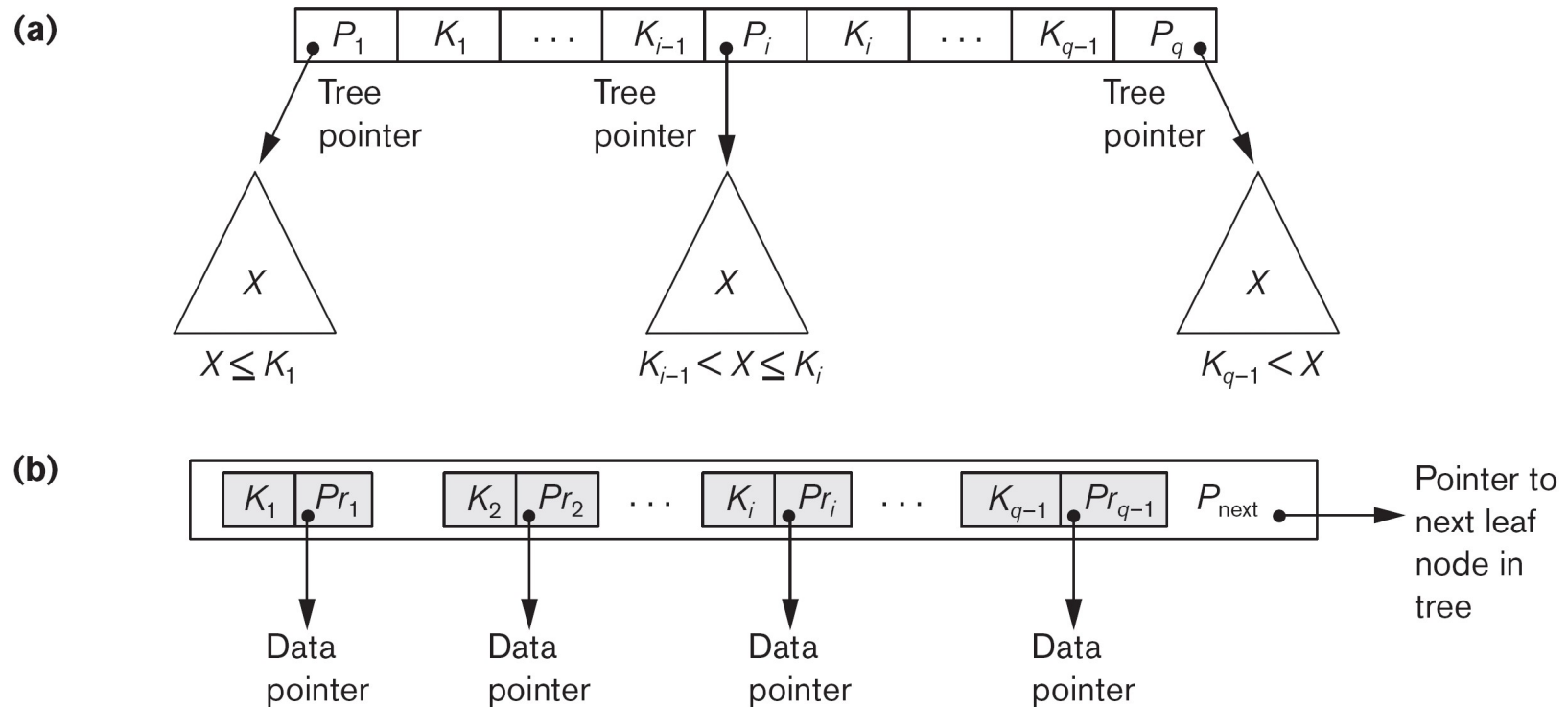
B-tree Index Insertion/Deletion

- Each B-tree node can have at most q tree pointers, $p - 1$ data pointers, and search key values
- Insertion
 - ◆ B-tree starts with a single root node at level 0
 - ◆ Once the root is full with $p - 1$ search key values and we attempt to insert another entry into the tree, the root node splits into two nodes at level 1
 - ◆ Only the middle value is kept in the root node and the rest of the values are split evenly between the other two nodes
 - ◆ When a non-root node is full and a new entry is inserted into it, that node is split into two nodes at the same level, and the **middle entry** is moved to the **parent node** along with **two pointers** to the new split nodes
 - ◆ If the parent node is full, it is also split

The Nodes of a B+-tree

Figure 18.11

The nodes of a B⁺-tree. (a) Internal node of a B⁺-tree with $q - 1$ search values.
 (b) Leaf node of a B⁺-tree with $q - 1$ search values and $q - 1$ data pointers.



B⁺-tree Index Insertion

- To find a place for the new key in the appropriate leaf, we put it there if there is room
- If a leaf node is full and new entry is inserted there, the node overflows
 - ◆ We **split** the leaf into two and divide the keys between the two new nodes so each is **half full**
 - ◆ The first $j = (p_{\text{leaf}} + 1)/2$ **entries** in the original node are kept while the remaining entries are moved to a new leaf node
- The **splitting of nodes** at one level appears to the level above as if a new key-pointer pair needs to be inserted at that higher level if there is room insert it. Otherwise **split the parent node** and continue up the tree

B⁺-tree Index Insertion

- The *j*th search value is replicated in the parent internal node and an extra pointer to the new node is created in the parent
- Note that every key value must exist at the leaf level
- Every value appearing in an internal node also appears as the (rightmost/leftmost) value in the leaf level of the subtree pointed at by the tree pointer to the left of the value
- If we try to insert into the root and there is no room, we split the root into two nodes and create a new root at the next higher level

B⁺-tree Index Deletion

- When an entry is deleted, it is always removed from the **leaf node**
- If it happens to occur in an internal node, it must also be removed and the value to its **left in the leaf node** must replace it in the internal node
- Deletion may cause underflow. Then, we try to find a **sibling leaf node** and redistribute the entries among the node and its sibling so that both are at least half full
- Otherwise, the node is merged with its sibling and the number of leaf node is reduced

Lec 10: Query Optimization

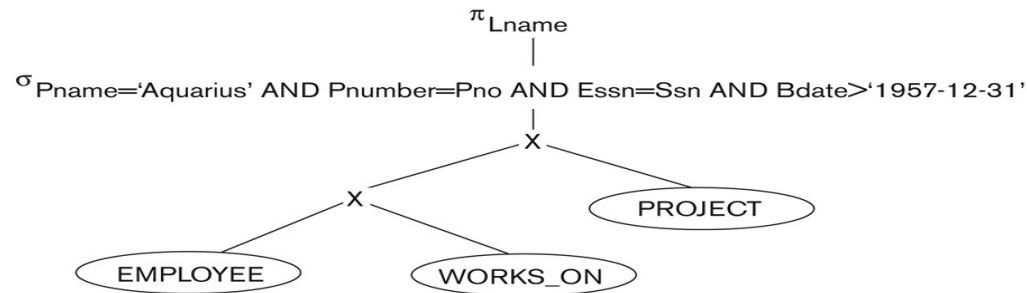
- How to implement Selection? S1-S9. Basic concepts to optimize Selection.
- How to implement Join?
 - Nested (inner-outer) loop** approach (brute force)
Using an **access structure to retrieve the matching record(s)**
 - Sort-merge join**
 - Hash-join**
- How to implement Aggregate Operation?

Using Heuristics in Query Optimization

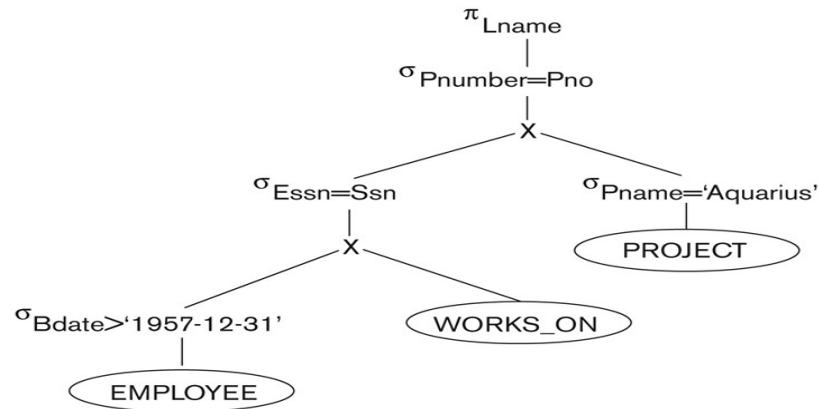
- Heuristic rule may be applied to modify the internal representation of a query (e.g., a query tree) to improve performance
- Process for heuristics optimization
Rule 1 – 12
- Heuristic rule may be applied to modify the internal representation of a query (e.g., a query tree) to improve performance
- By using the heuristic optimization, we can transform a query into a faster one, but there is no guarantee it is the best one.

Example

(a)



(b)



(c)

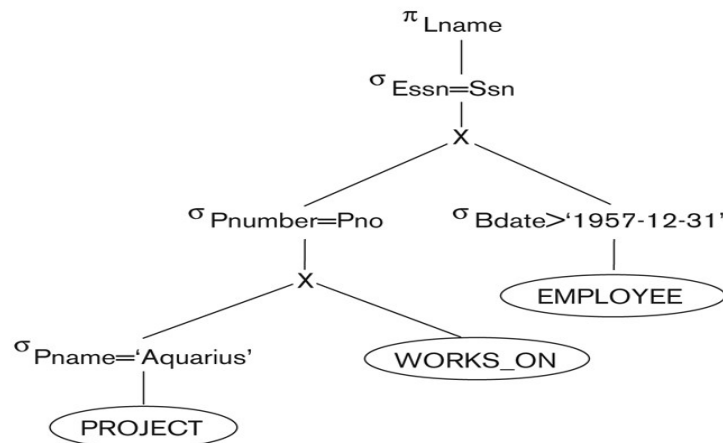


Figure 15.5

Steps in converting a query tree during heuristic optimization.

(a) Initial (canonical) query tree for SQL query Q.

(b) Moving SELECT operations down the query tree.

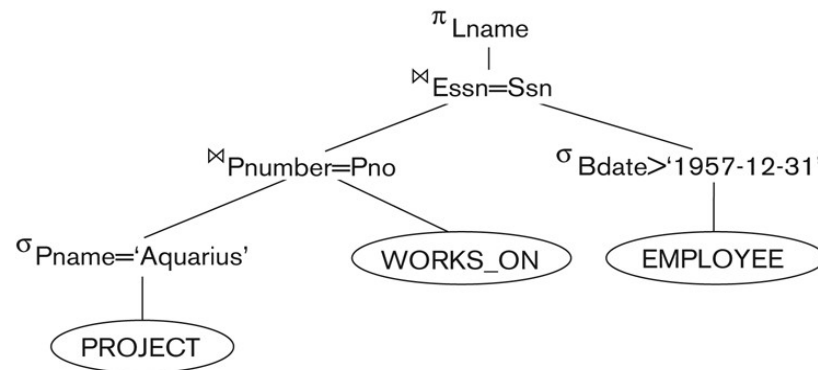
(c) Applying the more restrictive SELECT operation first.

(d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.

(e) Moving PROJECT operations down the query tree.

Example

(d)



(e)

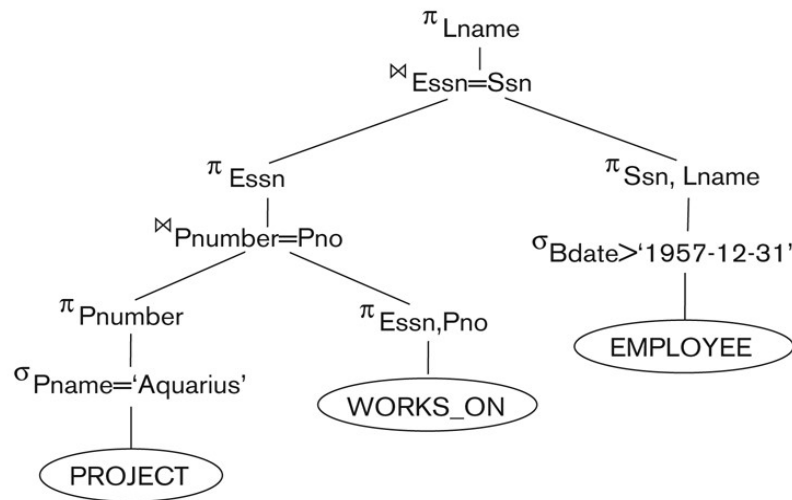


Figure 15.5

Steps in converting a query tree during heuristic optimization.

(a) Initial (canonical) query tree for SQL query Q.

(b) Moving SELECT operations down the query tree.

(c) Applying the more restrictive SELECT operation first.

(d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.

(e) Moving PROJECT operations down the query tree.

Summary of Heuristic Optimization

- The main heuristic is to first apply the operations that reduce the size of intermediate results.
 - ◆ This includes performing select operations as early as possible to reduce the number of tuples, and
 - ◆ performing project operations as early as possible to reduce the number of attributes. This is done by moving select and project operations as far down the tree as possible.
- In addition, the select and join operations that are most restrictive—that is, result in relations with the fewest tuples or with the smallest absolute size—should be executed before other similar operations.
 - ◆ This is done by reordering the leaf nodes of the tree among themselves and adjusting the rest of the tree appropriately.

Lec 11: Transaction

■ A Transaction:

- ◆ Logical unit of database processing that includes one or more access operations (read -retrieval, write - insert or update, delete).

■ Transaction boundaries:

- ◆ Begin and End transaction.

■ Transaction schedule

- ◆ The order of execution of **operations** from the transactions forms a **transaction schedule**

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

- A concurrent (non-serial) schedule:

a new transaction starts BEFORE the completion of the current transaction

- Serializable schedule:

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

Problem with Nonserial Schedule

Why Concurrency Control is needed?

■ **The Lost Update Problem**

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

■ **The Temporary Update (or Dirty Read) Problem**

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

■ **The Incorrect Summary Problem**

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

Schedules Classified on Serializability

- Result equivalent:

- ◆ Two schedules are called result equivalent if they produce the same final state of the database.

- Conflict equivalent:

- ◆ Two schedules are said to be conflict equivalent if the order of any two **conflicting operations** (RW,WW) is the same in both schedules.

- ***Conflict serializable:***

- ◆ A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S'.

Conflicting Operations

Operations of different transactions		Conflict	Reason
<i>read</i>	<i>read</i>	No	Because the effect of a pair of <i>read</i> operations does not depend on the order in which they are executed
<i>read</i>	<i>write</i>	Yes	Because the effect of a <i>read</i> and a <i>write</i> operation depends on the order of their execution
<i>write</i>	<i>write</i>	Yes	Because the effect of a pair of <i>write</i> operations depends on the order of their execution

Schedules Classified on Serializability

Testing for conflict serializability:

- ◆ Constructs a *precedence graph* (*serialization graph*) - a graph with directed edges
- ◆ An edge is created from T_i to T_j if one of the operations in T_i appears before a conflicting operation in T_j :
- ◆ W/R conflict: T_i executes $\text{write}(x)$ before T_j executes $\text{read}(x)$
- ◆ R/W conflict: T_i executes $\text{read}(x)$ before T_j executes $\text{write}(x)$
- ◆ W/W conflict: T_i executes $\text{write}(x)$ before T_j executes $\text{write}(x)$
- ◆ The schedule is serializable **if and only if** the precedence graph has no cycles.

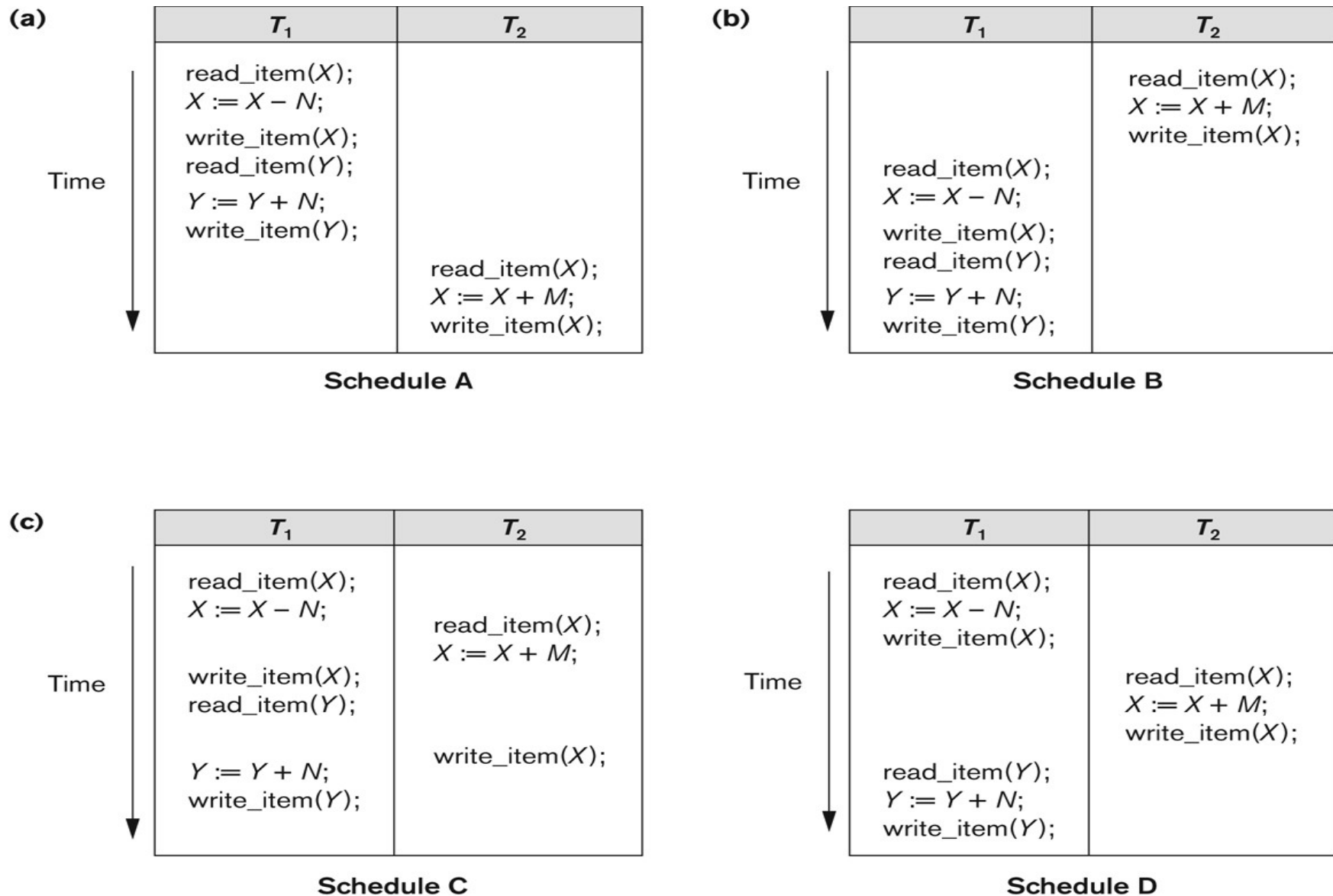
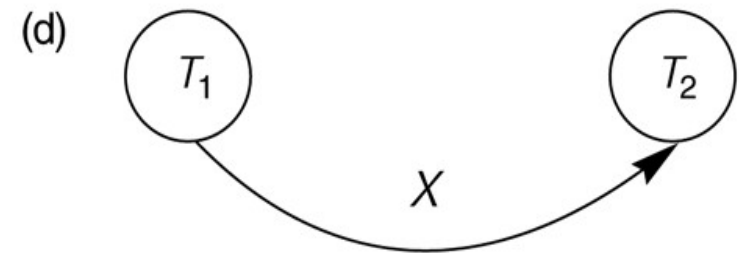
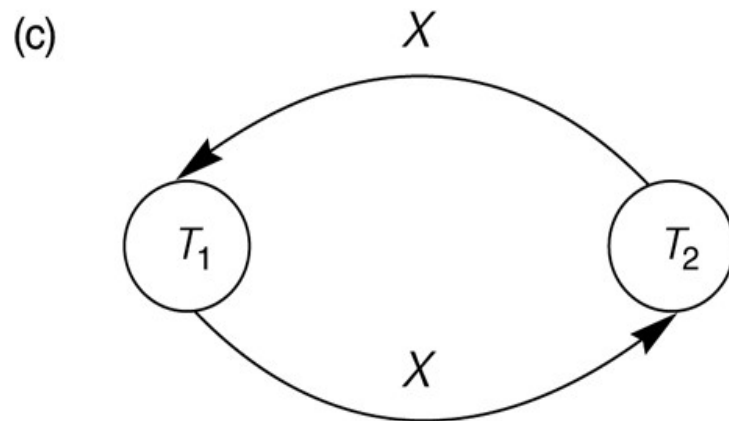
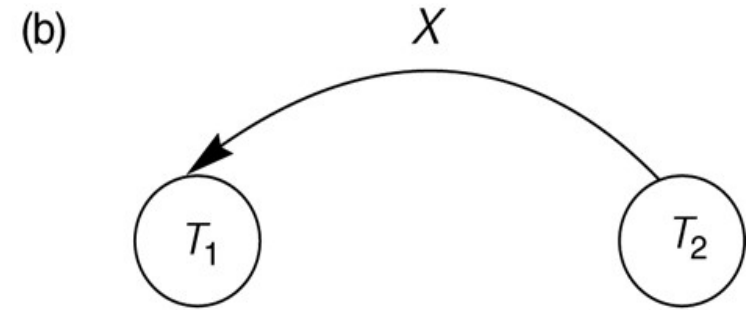
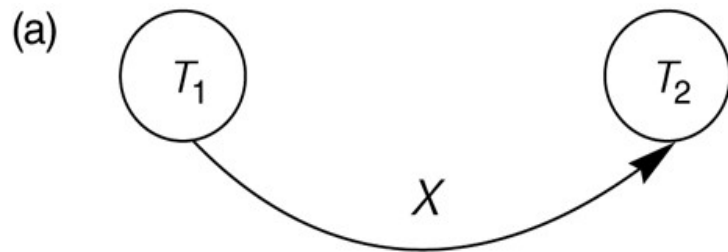


Figure 17.5

Examples of serial and nonserial schedules involving transactions T_1 and T_2 .
 (a) Serial schedule A: T_1 followed by T_2 . (b) Serial schedule B: T_2 followed by T_1 . (c) Two nonserial schedules C and D with interleaving of operations.



- Constructing the precedence graphs for schedules A and D from Figure 17.5 to test for conflict serializability.
 - ◆ (a) Precedence graph for serial schedule A.
 - ◆ (b) Precedence graph for serial schedule B.
 - ◆ (c) Precedence graph for schedule C (not serializable).
 - ◆ (d) Precedence graph for schedule D (serializable, equivalent to schedule A).

Schedules Classified on Recoverability

- **Recoverable schedule:**

- ◆ No committed transaction needs to be rolled back.

- **A schedule S is recoverable if**

- ◆ no transaction T in S commits until all transactions T' that have written some item X that T reads have committed.

- **Cascaded rollback:**

- ◆ A single failure leads to a series of rollback
- ◆ All uncommitted transactions that read an item from a failed transaction must be rolled back.

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.

Lec 11: Concurrency Control

- Purposes of Concurrency Control
 - ◆ To preserve database consistency
 - ◆ To ensure all schedules are serializable
 - ◆ To maximize the system performance (higher concurrency)

Lock

- Lock Manager:
 - ◆ Managing locks on data items
- Lock table:
 - ◆ An array to show the lock entry for each data item
 - ◆ Each entry of the array stores the identify of transaction that has set a lock on the data item including the mode
- One entry for one database? One record? One field?
Lock granularity (Coarse granularity vs. fine granularity)
- Larger granularity -> higher conflict probability but lower locking overhead
- Note: Lock and Unlock are atomic operations

Lock

■ Binary Locks

- ◆ two states or values: locked=1 and unlocked=0
- ◆ lock(X): set the value to be 1, item X cannot be accessed by other transaction
- ◆ unlock(X): set the value to be 0, item X can be accessed by other transaction

Lock

- Shared/Exclusive (or Read/Write) Locks

- ◆ `read_lock(X)`: set to be the share state, more than one transaction can apply shared lock on X for reading its value but no write lock can be applied on X by any other transaction.

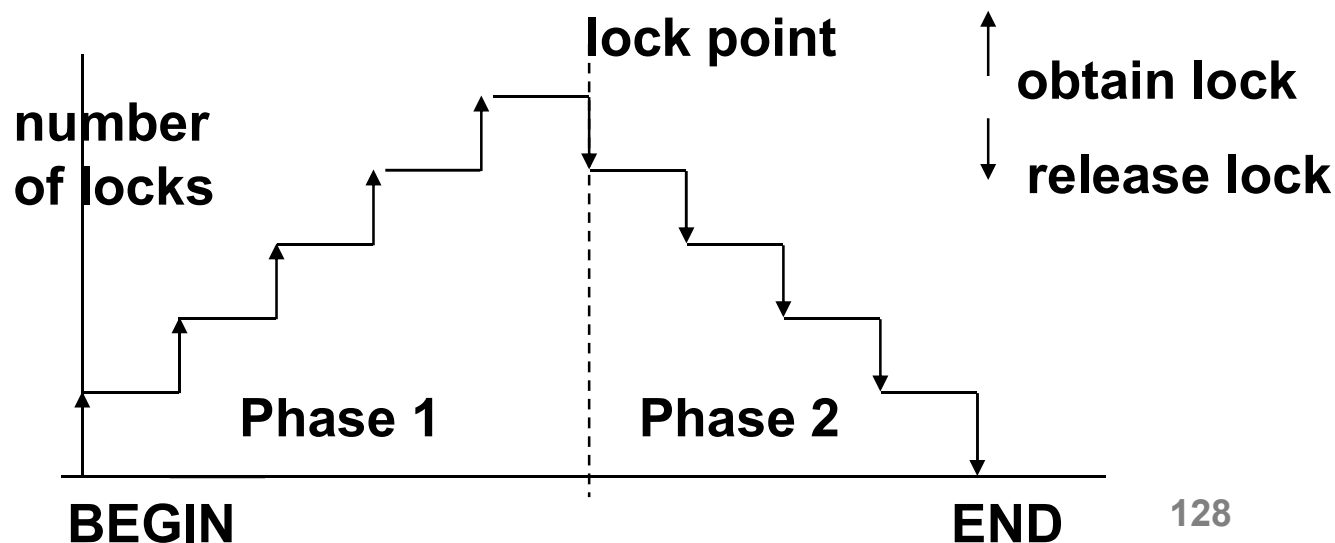
- ◆ `write_lock(X)`: set to be the exclusive state, only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X.

- ◆ `unlock(x)`: set to be unlock state, item X can be accessed by other transaction

	Read	Write
Read	Y	N
Write	N	N

Basic Two Phase Locking (B2PL)

- All locking operations precede the first unlock operation in the transaction.
- A transaction can be divided into two phases: **growing phase** and **shrinking phase**
- It can guarantee that all pairs of conflicting operations of two transactions are scheduled in **the same order**. -> guarantee serializability
 - E.g., $T1 \rightarrow T2$ or $T2 \rightarrow T1$ and NO $T1 \leftrightarrow T2$



Conservative Two Phase Locking (C2PL)

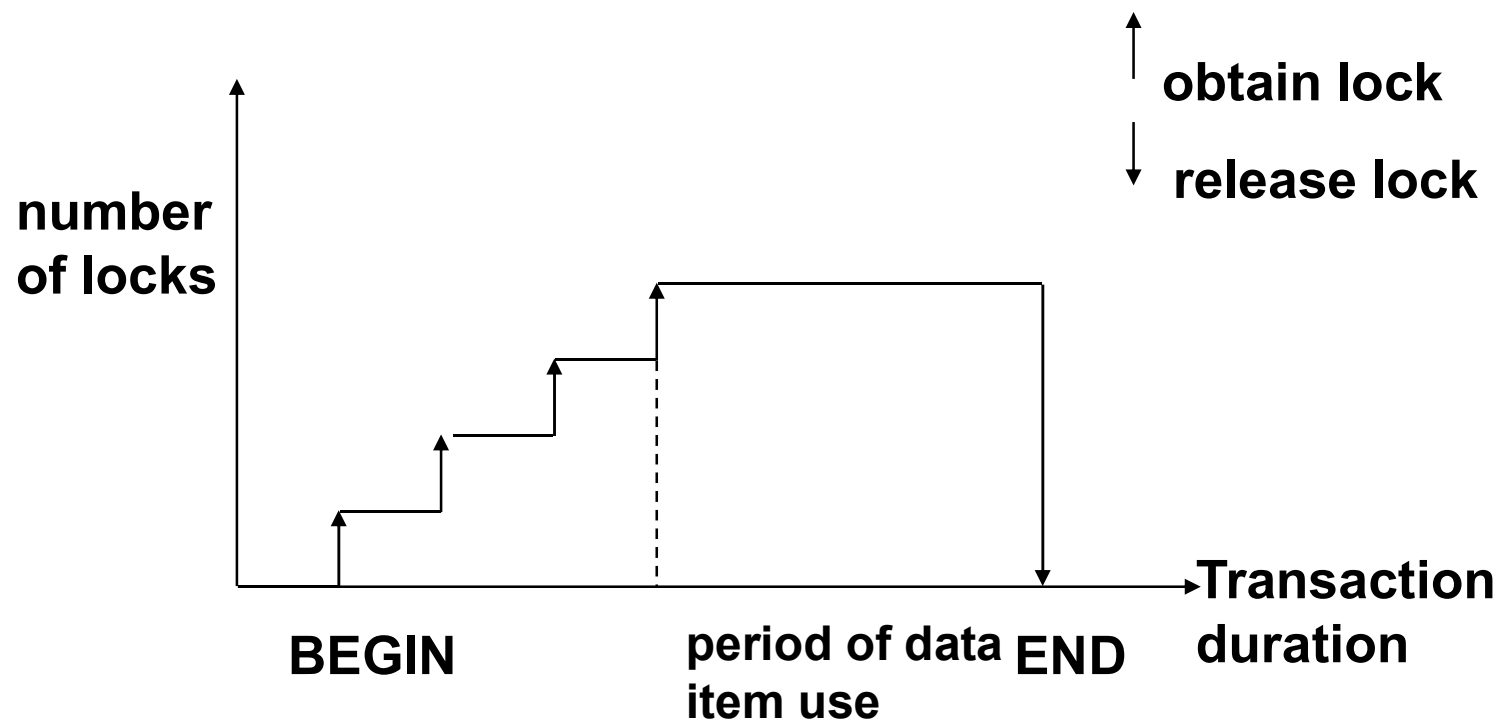
- requires a transaction to lock all the items it accesses before the transaction begins execution.
- If any of the predeclared items needed cannot be locked, the transaction does not lock any item; instead, it waits until all the items are available for locking.
- Every time the scheduler releases the locks of a complete transactions, it examines the waiting queue to see if it can grant all the lock requests of any waiting transactions
- 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)

129

Strict Two Phase Locking (S2PL)

- Requires a transaction T does not release any of its exclusive (write) locks until after it commits or aborts. Hence, no other transaction can read or write an item that is written by T unless T has committed.
- Compared with B2PL, the lock holding time may be longer and the concurrency may be lower
- Compared with C2PL, the lock holding time may be shorter and the concurrency may be higher
- It may have the **problem of deadlock** but all schedule is recoverable

Strict Two Phase Locking (S2PL)



Deadlock

T1

read_lock (Y);
read_item (Y);

write_lock (X);
(waits for X)

T2

read_lock (X);
read_item (X);

write_lock (Y);
(waits for Y)

T1 and T2 do follow two-phase policy but they are deadlock

◆ Deadlock (T1 and T2)

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.

Deadlock Prevention

- Deadlock condition
 - ◆ Hold and wait
 - ◆ Cyclic wait
- Deadlock Prevention
 - 1 A transaction locks all data items it refers to before it begins execution (C2PL). This way of locking prevents deadlock since a transaction never waits for a data item
 - 2 Ordering all the items in the database and making sure that a transaction that needs several items will lock them according to that order
- 3 Timestamp: Each transaction is assigned a unique **time-stamp**.
Two Schemes preventing deadlock:
- Wait-die and Wound-Wait

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.
- ◆ A wait-for-graph is created using the lock table. As soon as a transaction is blocked, it is added to the graph. When a chain like: T_i waits for T_j , T_j waits for T_k and T_k waits for T_i occurs, this creates a cycle. One of the transactions will be chosen to abort.

Summary

1. Know the desirable properties of transactions
2. Understand the difference among schedules based on **Recoverability** (non-recoverable, recoverable, cascadeless, and strict)
3. Understand the difference among schedules based on **Serializability** (conflict serializable and non-serializable)
4. Know the purpose of concurrency control and able to judge whether a schedule satisfies Two-Phase locking (2PL)!