



# Part 1: Overview of Databases

Kulsawasd Jitkajornwanich, PhD

kulsawasd.ji@kmitl.ac.th

[Taken and adapted from slides and/or other materials by Ramez Elmasri and Shamkant B. Navathe,  
“Fundamentals of Database Systems”, Addison-Wesley, Pearson]



# Chapter 1: Database System Concepts and Architecture



คณะวิทยาศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง  
Faculty of Science, King Mongkut's Institute of Technology Ladkrabang

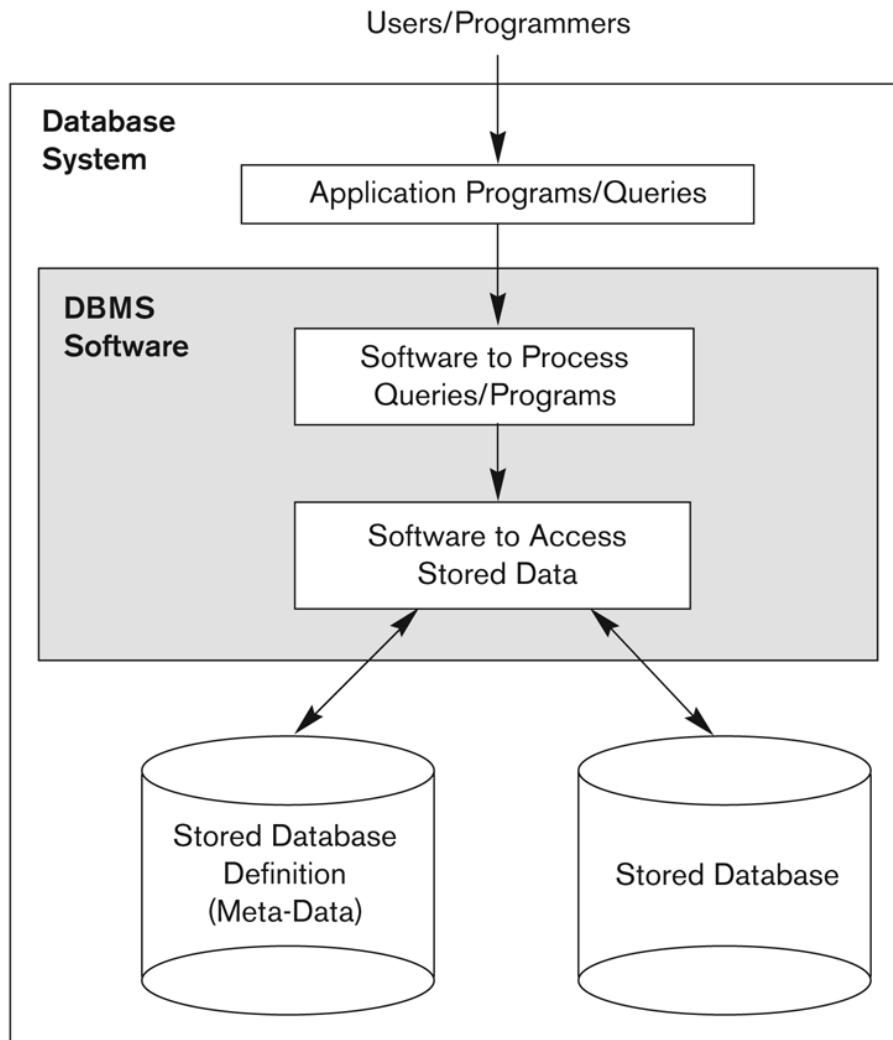
# Outline

- Basic Definitions
- Typical DBMS Functionality
- Main Characteristics of the Database Approach
- Advantages of Using the Database Approach
- When Not to Use Databases
- Data Models and Their Categories
- Three-Schema Architecture and Data Independence
- Database System Architectures
- Classification of DBMSs

# Basic Definitions

- **Database:**
  - A collection of related data.
- **Data:**
  - Known facts that can be recorded and have an implicit meaning.
- **Mini-world:**
  - Some part of the real world about which data is stored in a database. For example, student registration, grades and transcripts at a university.
- **Database Management System (DBMS):**
  - A software package/system to facilitate the creation and maintenance of a computerized database.
- **Database System:**
  - The DBMS software together with the data itself. Sometimes, the application programs and interfaces are also included.

# Simplified database system environment (Figure 1.1)



## RELATIONS

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

## COLUMNS

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

Note: Major\_type is defined as an enumerated type with all known majors. XXXXNNNN is used to define a type with four alpha characters followed by four digits

**Figure 1.3**

An example of a database catalog for the database in Figure 1.2.

**Figure 1.1**

A simplified database system environment.

# Typical DBMS Functionality

- *Defining* a particular database in terms of its data types, structures, and constraints
- *Manipulating* the database:
  - Modification: Insertions, deletions and updates to its content
  - Retrieval: Querying, generating reports
  - Accessing/changing the database through Web applications
- *Processing* and *Sharing* by a set of concurrent users and application programs – yet, keeping all data valid and consistent

# Typical DBMS Functionality

- Other features:
  - Protection or Security measures to prevent unauthorized access
  - “Active” processing to take internal actions on data
  - Presentation and Visualization of data
  - Maintaining the database and associated programs over the lifetime of the database application
  - Called database, software, and system life-cycle maintenance

# Example of a Database UNIVERSITY Application

- **Mini-world for the example:**
  - Part of a UNIVERSITY environment.
- **Some mini-world entities:**
  - STUDENTs
  - COURSEs
  - SECTIONs (of COURSEs)
  - (academic) DEPARTMENTs
  - INSTRUCTORs
- **Some mini-world relationships:**
  - SECTIONs *are of specific* COURSEs
  - STUDENTs *take* SECTIONs
  - COURSEs *have prerequisite* COURSEs
  - INSTRUCTORs *teach* SECTIONs
  - COURSEs *are offered by* DEPARTMENTs
  - STUDENTs *major in* DEPARTMENTs

# Example of a Database UNIVERSITY Application

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

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

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

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

PREREQUISITE	
Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

**Figure 1.2**

A database that stores student and course information.

# Main Characteristics of the Database Approach

- **Self-describing nature of a database system:**
  - A DBMS **catalog** stores the *description* of a particular database (e.g. data structures, types, and constraints)
  - The description is called **meta-data**
  - This allows the DBMS software to work with *different* database applications (university, bank, airlines, etc.)
- **Insulation between programs and data** (called, program-data independence):
  - Allows changing data structures and data storage organization without having to change the DBMS access programs (hide storage details). (e.g., adding a new column to a table/index)
- **Support of multiple views of the data:**
  - Each user may see a different view of the database, which describes **only** the data of interest to that user.

# Main Characteristics of the Database Approach

- **Sharing of data and multi-user transaction processing:**
  - Allowing a set of **user transactions** to access and update the database *concurrently* (at the same time).
  - *Concurrency control* within the DBMS guarantees that each **transaction** is correctly executed or aborted
  - *Recovery subsystem* ensures each completed transaction has its effect permanently recorded in the database
  - **OLTP** (Online Transaction Processing) is a major part of database applications (allows hundreds of concurrent transactions to execute per second)

# Types of Database Users

- Users may be divided into
  - Those who actually use and control the database content, and those who design, develop and maintain database applications (called “Actors on the Scene”), and
  - Those who design and develop the DBMS software and related tools, and the computer systems operators (called “Workers Behind the Scene”).

# Database Users: End-users

- Actors on the scene
  - **End-users:** Use the database for queries, reports, and updating the database content. Can be categorized into:
    - **Casual end-users:** access database occasionally when needed
    - **Naïve (or Parametric) end-users:** largest section of end-user population.
      - Use previously implemented and tested programs (called “canned transactions”) to access/update the database.
      - Examples are bank-tellers or hotel reservation clerks or sales clerks.

# Database Users: End-users (cont)

- **Sophisticated end-users:**
  - These include business analysts, scientists, engineers, others thoroughly familiar with the system capabilities.
  - Many use tools in the form of software packages that work closely with the stored database.
- **Stand-alone end-users:**
  - Mostly maintain personal databases using ready-to-use packaged applications.
  - An example is a tax program user that creates its own internal database.
  - Another example is a user that maintains an address book

# Database Users: DB Administrators (DBAs) and Designers

- Actors on the scene (continued)
  - **Database administrators:**
    - Responsible for authorizing/controlling access to the database; coordinating and monitoring its use; acquiring software and hardware resources; and monitoring efficiency of operations.
  - **Database Designers:**
    - Responsible for defining database structure, constraints, and transactions; communicate with users to understand their needs.

# Advantages of Using the Database Approach

- Controlling redundancy in data storage and in development and maintenance efforts.
  - Sharing of data among multiple users.
- Restricting unauthorized access to data.
- Providing persistent storage for program Objects
  - In Object-oriented DBMSs
- Providing Storage Structures (e.g. indexes) for efficient Query Processing

# Advantages of Using the Database Approach (cont.)

- Providing backup and recovery services.
- Providing multiple interfaces to different classes of users.
- Representing complex relationships among data.
- Enforcing integrity constraints on the database.
- Drawing inferences and actions from the stored data using deductive and active rules
- Allowing multiple “views” of the same data (see next slide)

## TRANSCRIPT

Student_name	Student_transcript				
	Course_number	Grade	Semester	Year	Section_id
Smith	CS1310	C	Fall	08	119
	MATH2410	B	Fall	08	112
Brown	MATH2410	A	Fall	07	85
	CS1310	A	Fall	07	92
	CS3320	B	Spring	08	102
	CS3380	A	Fall	08	135

(a)

## COURSE\_PREREQUISITES

Course_name	Course_number	Prerequisites
Database	CS3380	CS3320
		MATH2410
Data Structures	CS3320	CS1310

(b)

### Figure 1.5

Two views derived from the database in Figure 1.2. (a) The TRANSCRIPT view.

(b) The COURSE\_PREREQUISITES view.

# Additional Implications of Using the Database Approach

- Potential for enforcing standards:
  - Crucial for the success of database applications in large organizations. **Standards** refer to data item names, display formats, screens, report structures, meta-data (description of data), Web page layouts, etc.
- Reduced application development time:
  - The time needed to add each new application is reduced.

# Additional Implications of Using the Database Approach (cont.)

- Flexibility to change data storage structures:
  - Storage structures may evolve to improve performance, or because of new requirements.
- Availability of up-to-date information:
  - Extremely important for on-line transaction systems such as airline, hotel, car reservations.
- Economies of scale:
  - Wasteful overlap of resources and personnel can be avoided by consolidating data and applications across departments.

# Historical Development of Database Technology

- Early Database Applications:
  - The Hierarchical and Network Models were introduced in mid 1960s and dominated during the seventies.
  - Some worldwide database processing still occurs using these models; particularly, the hierarchical model.
- Relational Model based Systems:
  - Relational model was introduced in 1970, and heavily researched and experimented with at IBM Research and several universities.
  - Relational DBMS Products emerged in the early 1980s and now dominate the market.

# Historical Development of Database Technology

- Object-oriented and emerging applications:
  - Object Databases (ODBs) were introduced in late 1980s and early 1990s to cater to the need of complex data and applications, and the proliferation of object-oriented programming languages.
    - Their use has not taken off much.
  - Many relational DBMSs have incorporated object database concepts, leading to a new category called *object-relational* databases (ORDBs)
  - *Extended relational* systems add further capabilities (e.g. for multimedia data, XML, spatial, and other data types)

# Historical Development of Database Technology

- Data on the Web and E-commerce Applications:
  - *Static* Web pages often specified in HTML (Hypertext markup language) with links among pages.
  - *Dynamic* Web pages have portions of their content extracted from databases, and allow user interaction with databases by typing in form boxes.
  - Script programming languages such as PHP and JavaScript allow generation of dynamic Web pages, and provide for user querying of databases by typing selection keywords (e.g. flight number or student id).
  - Also allow database updates through Web pages

# Extending Database Capabilities

- New functionality is being added to DBMSs in the following areas:
  - Scientific Applications
  - XML (eXtensible Markup Language)
  - Image Storage and Management
  - Audio and Video Data Management
  - Data Warehousing and Data Mining
  - Spatial Data Management and Geographic Information Systems
  - Time Series and Historical Data Management
  - Collecting and fusing data from distributed sensors
- The above led to *new research and development* in incorporating new data types, complex data structures, new operations/query languages, and new storage and indexing schemes.

# When not to use a DBMS

- Main inhibitors (costs) of using a DBMS:
  - High initial investment and possible need for additional hardware.
  - Overhead for providing generality, security, concurrency control, recovery, and other functions.
- When a DBMS may be unnecessary:
  - If the database and applications are simple, well defined, and not expected to change.
  - If there are stringent real-time requirements that may not be met because of DBMS overhead.
  - If access to data by multiple users is not required.

# When not to use a DBMS (cont.)

- When no DBMS may suffice:
  - If the database system is not able to handle the complexity of data because of modeling limitations
  - If the database users need special operations not supported by the DBMS
  - When DBMS overhead makes it impossible to achieve the needed application performance

# Database Schema versus Database State

- Database Schema:
  - The *description* of a database.
  - Includes descriptions of the database structure, relationships, data types, and constraints
- Schema Diagram:
  - An *illustrative* display of (some aspects of) a database schema
- Schema Construct:
  - A *component* of the schema or an object within the schema, e.g., STUDENT, COURSE, Name

# Database Schema vs. Database State (cont.)

- Database State:
  - The actual data stored in a database at a *particular moment in time*. This includes the collection of all the data in the database.
  - Also called a database instance (or occurrence or snapshot).
    - NOTE: The term *instance* is also used to refer to individual database components, e.g. a *record instance*, *table instance*, or *entity instance*

# Database Schema vs. Database State

- Database State:
  - Refers to the *content* of a database at a particular moment in time.
- Initial Database State:
  - Refers to the database state when it is initially loaded into the system.
- Valid State:
  - A state that satisfies the structure and constraints of the database.

# Database Schema vs. Database State (cont.)

- Distinction
  - The *database schema* changes very infrequently.
  - The *database state* changes every time the database is updated.
- **Schema** is also called **intension**.
- **State** is also called **extension**.

# Example of a Database Schema

## STUDENT

Name	Student_number	Class	Major
------	----------------	-------	-------

## COURSE

Course_name	Course_number	Credit_hours	Department
-------------	---------------	--------------	------------

## PREREQUISITE

Course_number	Prerequisite_number
---------------	---------------------

## SECTION

Section_identifier	Course_number	Semester	Year	Instructor
--------------------	---------------	----------	------	------------

## GRADE\_REPORT

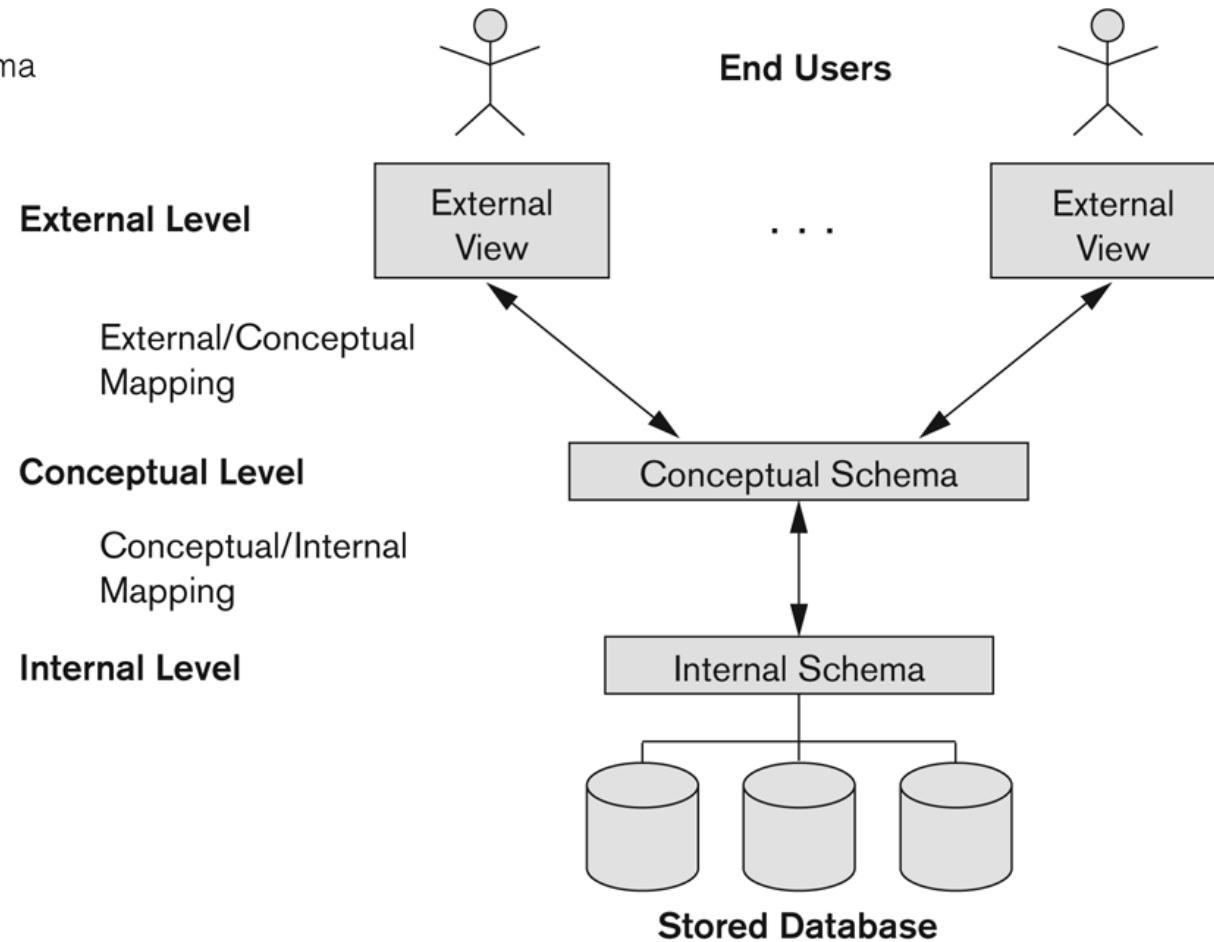
Student_number	Section_identifier	Grade
----------------	--------------------	-------

**Figure 2.1**

Schema diagram for the database in Figure 1.2.

# Three-Schema Architecture

**Figure 2.2**  
The three-schema architecture.



# Three-Schema Architecture (cont.)

- Defines DBMS schemas at *three* levels:
  - **Internal schema** at the internal level to describe physical storage structures and access paths (e.g., indexes).
  - **Conceptual schema** at the conceptual level to describe the structure and constraints for the whole database for a community of users.
  - **External schemas** at the external level to describe the various user views.

# Three-Schema Architecture (cont.)

- Mappings among schema levels are needed to transform requests and data.
  - Users and programs refer to an external schema, and are mapped by the DBMS to the internal schema for execution.
  - Data extracted from the internal DBMS level is reformatted to match the user's external view (e.g. formatting the results of an SQL query for display as a Web page)

# Data Independence

- **Logical Data Independence:**
  - The capacity to change the *conceptual* schema without having to change the *external* schemas and their associated application programs.
  - E.g., add a new column to the GRADE table, grade report Web site remains unchanged
- **Physical Data Independence:**
  - The capacity to change the *internal* schema without having to change the *conceptual* schema.
  - For example, the internal schema may be changed when certain file structures are reorganized or new indexes are created to improve database performance
  - E.g., change an index structure for a table, the same query still works but faster

# Data Independence (cont.)

- When a schema at a lower level is changed, only the **mappings** between this schema and higher-level schemas need to be changed in a DBMS that fully supports data independence.
- The higher-level schemas themselves are **unchanged**.
  - Hence, the application programs need not be changed since they refer to the external schemas.

# DBMS Languages

- **Data Definition Language (DDL):**
  - Used by the DBA and database designers to specify the conceptual schema of a database.
  - In many DBMSs, the DDL is also used to define internal and external schemas (views).

# DBMS Languages (cont.)

- **Data Manipulation Language (DML):**
  - Used to specify database retrievals and updates
  - DML commands (data sublanguage) can be *embedded* in a general-purpose programming language (host language), such as COBOL, C, C++, or Java
    - A library of functions can also be provided to access the DBMS from a programming language
  - Alternatively, stand-alone DML commands can be applied directly (called a *query language*).

# DBMS Interfaces

- Stand-alone query language interfaces
  - Example: Typing SQL queries directly through the DBMS interactive SQL interface (e.g. SQL\*Plus in ORACLE)
- Programmer interfaces for embedding DML in programming languages
- User-friendly interfaces (often Web-based)
  - Menu-based, forms-based, graphics-based, etc.

# DBMS Programming Language Interfaces

- Programmer interfaces for embedding DML in a programming language (see Chapter 13):
  - **Embedded Approach:** e.g., embedded SQL (EXEC SQL) (for C, C++, etc.), SQLJ (#sql) (for Java), allows applications programmers to embed SQL statements in Java code.
  - **Procedure Call Approach:** (API) e.g. JDBC class library for Java, ODBC for other programming languages
  - **Database Programming Language Approach:** e.g. ORACLE has PL/SQL, a programming language based on SQL; language incorporates SQL and its data types as integral components

# Typical DBMS Component Modules

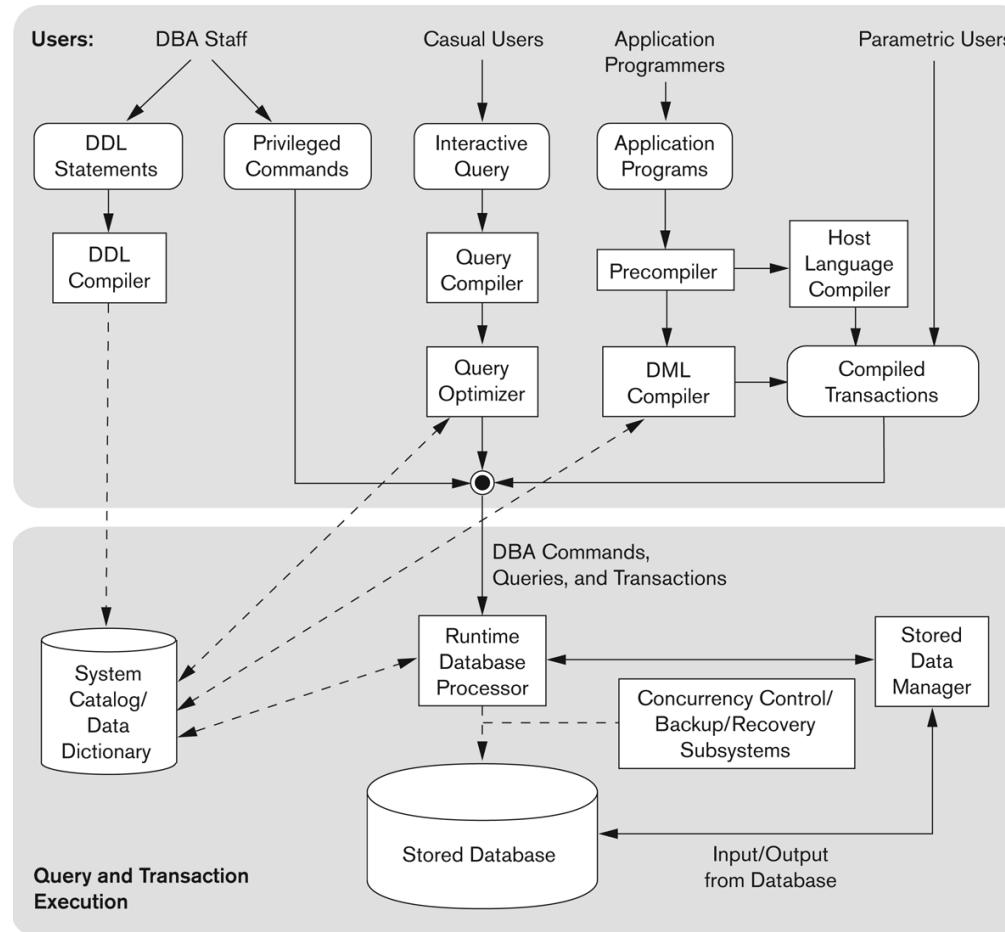


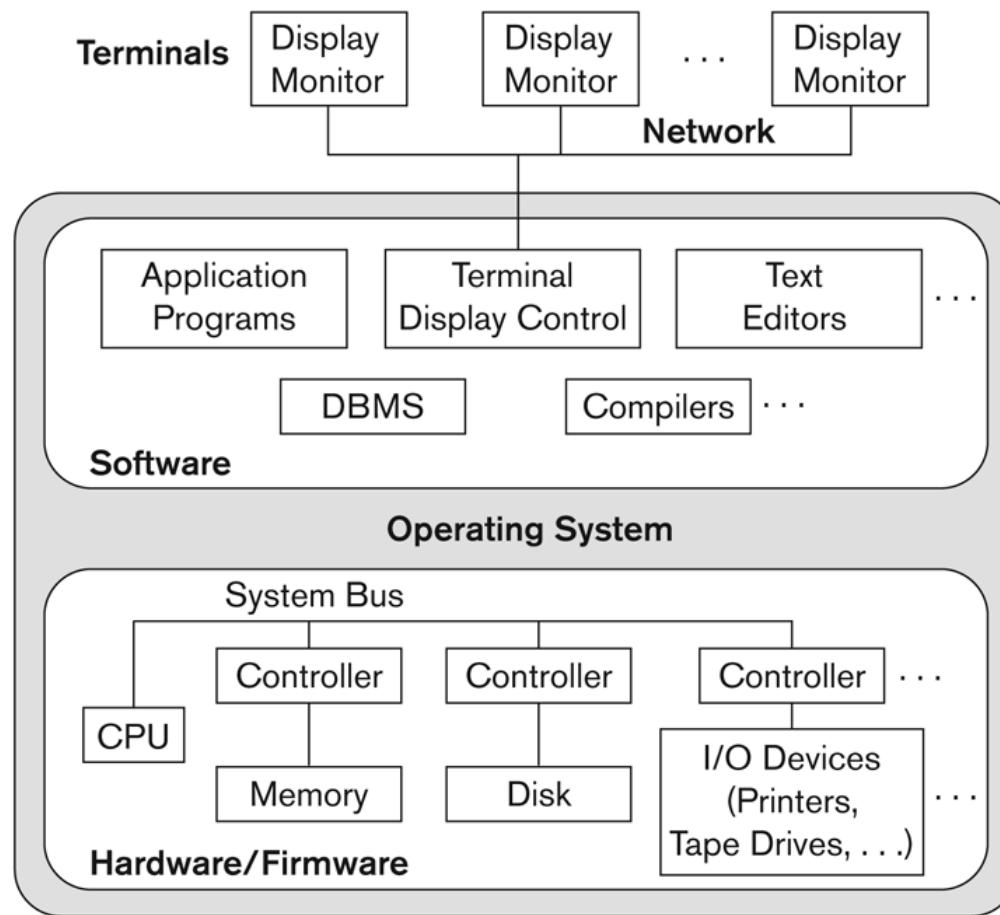
Figure 2.3

Component modules of a DBMS and their interactions.

# DBMS Architectures

- Centralized DBMS Architecture:
  - Combines everything into single computer system, including: DBMS software, hardware, application programs, and user interface processing software.
  - User can still connect through a remote terminal – however, all processing is done at centralized site (computer).

# A Physical Centralized Architecture



**Figure 2.4**

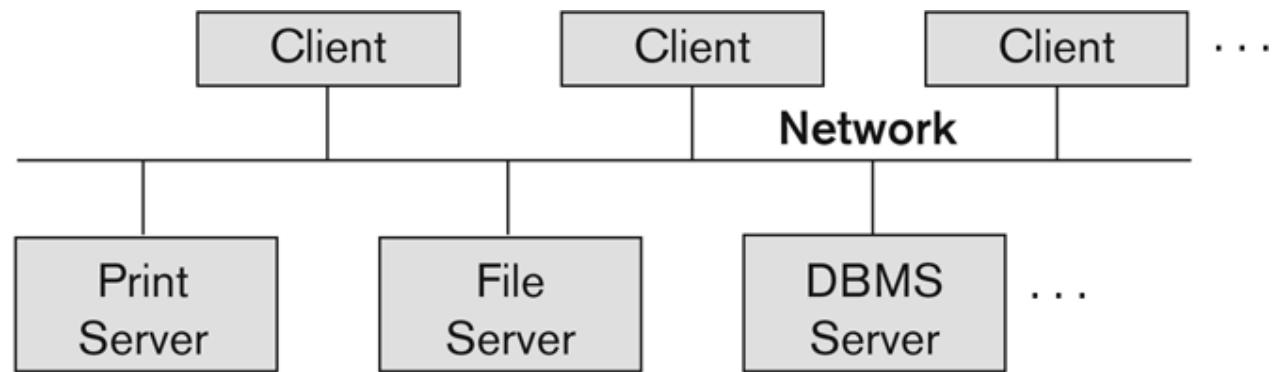
A physical centralized architecture.

# DBMS Architectures (cont.)

- Basic 2-tier Client-Server Architecture: Specialized Server nodes with Specialized functions
  - Print server
  - File server
  - DBMS server
  - Web server
  - Email server
- Client nodes can access the specialized servers as needed

# Logical two-tier client server architecture

**Figure 2.5**  
Logical two-tier  
client/server  
architecture.



# Client nodes

- Provide appropriate interfaces through a client software module to access and utilize the various server resources.
- Clients may be PCs or Workstations (or even diskless machines) with the client software installed.
- Connected to the servers via a network.
  - LAN: local area network
  - wireless network
  - etc.

# DBMS Server

- Provides database query and transaction services to the clients
- Relational DBMS servers are often called SQL servers, query servers, or transaction servers
- Applications running on clients utilize an Application Program Interface (**API**) to access server databases via standard interface such as:
  - ODBC: Open Database Connectivity standard
  - JDBC: for Java programming access
- Client and server must install appropriate client module and server module software for ODBC or JDBC

# Two Tier Client-Server DBMS Architecture

- A program running at a client may connect to several DBMSs (also called data sources).
- In general, data sources can be files or other non-DBMS software that manages data.
- Client focuses on user interface interactions and only accesses database when needed.
- In some cases (e.g. some object DBMSs), more functionality is transferred to clients (e.g. data dictionary functions, optimization and recovery across multiple servers, etc.)

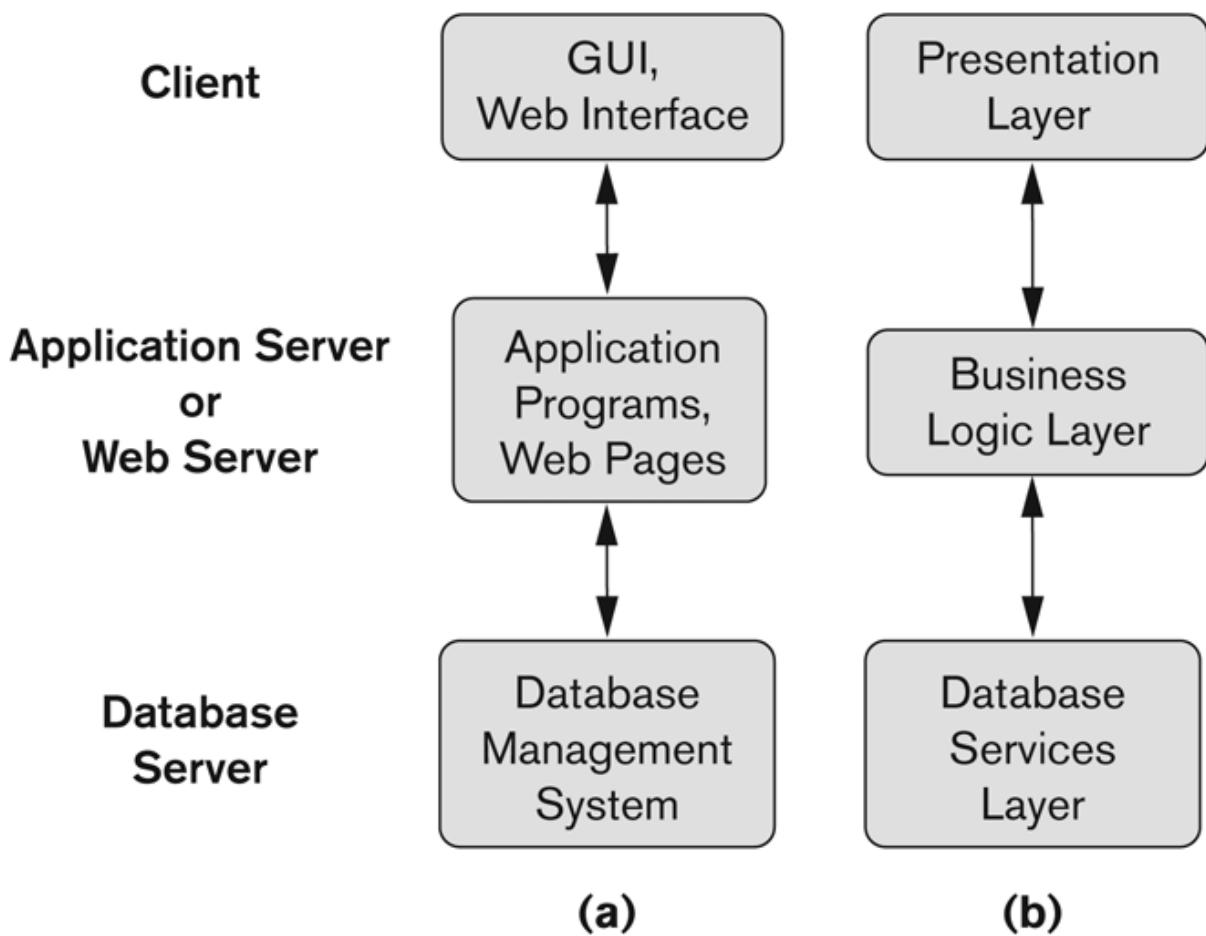
# Three Tier Client-Server DBMS Architecture

- Common for Web applications
- Third intermediate layer (middle tier) called Application Server or Web Server:
  - Stores the web connectivity software and the business logic part of the application
  - Accesses and updates data on the database server
  - Acts like a conduit for sending partially processed data between the database server and the client.
- Three-tier Architecture Can Enhance Security:
  - Database server only accessible via middle tier
  - Clients cannot directly access database server

# Three-tier client-server architecture

**Figure 2.7**

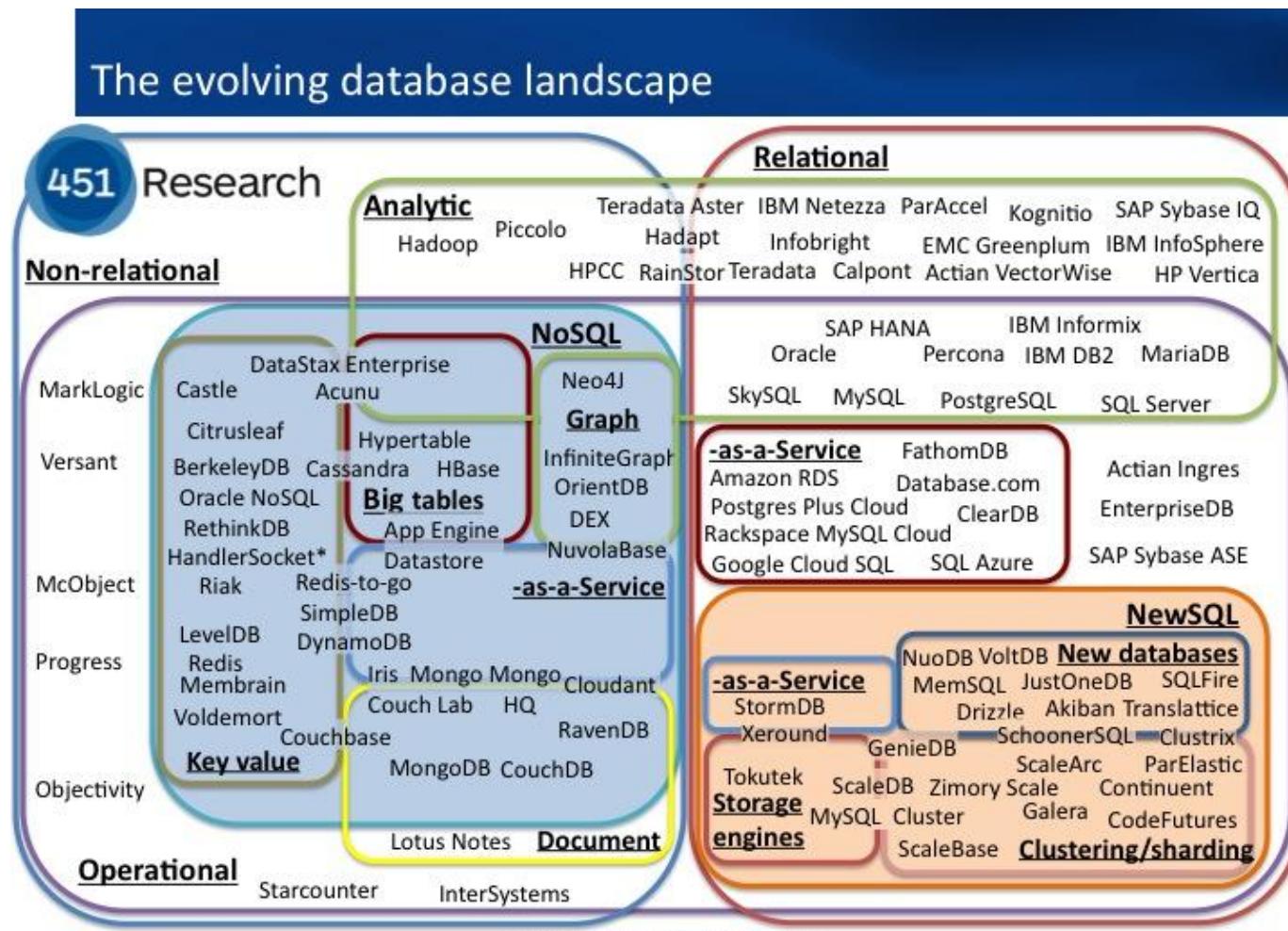
Logical three-tier client/server architecture, with a couple of commonly used nomenclatures.



# Classification of DBMSs

- Based on the data model used
  - Traditional: Relational, Network, Hierarchical, Object-oriented, Object-relational.
  - Emerging: NoSQL, NewSQL
- Other classifications
  - Single-user (typically used with personal computers) vs. multi-user (most DBMSs).
  - Centralized (uses a single computer with one database) vs. distributed (uses multiple computers, multiple databases)

# Classification of DBMSs



© 2012 by The 451 Group. All rights reserved

# Cost considerations for DBMSs

- Cost Range: from free open-source systems to configurations costing millions of dollars
- Examples of free relational DBMSs: MySQL, PostgreSQL, others
- Commercial DBMSs offer additional specialized modules, e.g. time-series module, spatial data module, document module, XML module
  - These offer additional specialized functionality when purchased separately
  - Sometimes called cartridges (e.g., in Oracle) or blades
- Different licensing options: site license, maximum number of concurrent users (seat license), single user, etc.



# Part 2: Relational Databases

Kulsawasd Jitkajornwanich, PhD

kulsawasd.ji@kmitl.ac.th

[Taken and adapted from slides and/or other materials by Ramez Elmasri and Shamkant B. Navathe,  
“Fundamentals of Database Systems”, Addison-Wesley, Pearson]



# Chapter 2: The Relational Data Model and Relational Database Constraints



คณะวิทยาศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง  
Faculty of Science, King Mongkut's Institute of Technology Ladkrabang

# Outline

- Relational Model Concepts
- Relational Model Constraints and Relational Database Schemas
- Update Operations and Dealing with Constraint Violations

# Relational Model Concepts

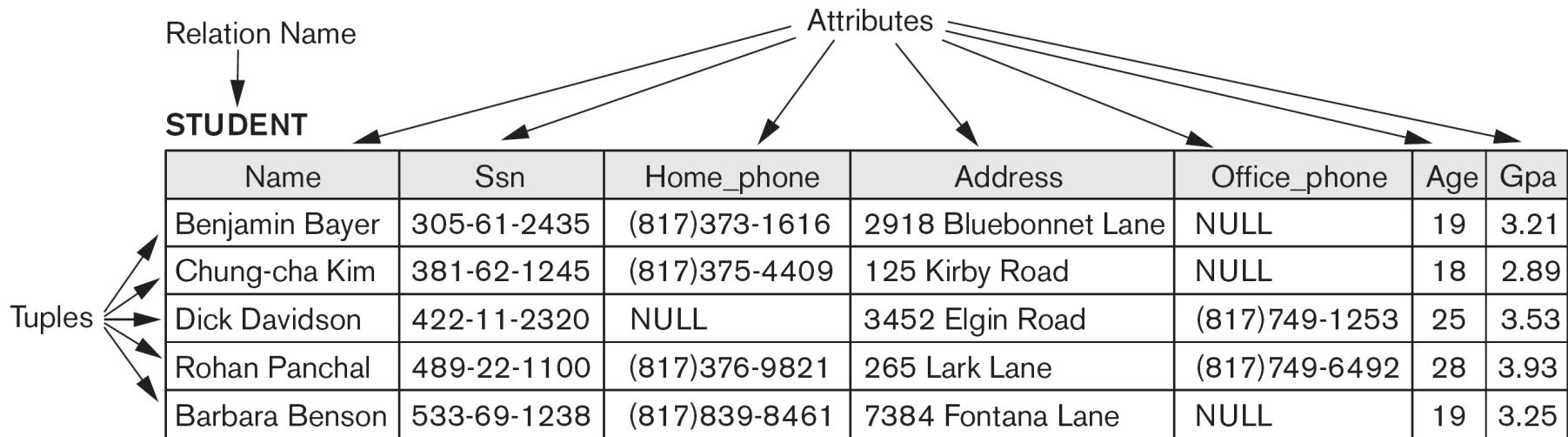
- The relational Model of Data is based on the concept of a *Relation*
  - Has a formal mathematical foundation provided by set theory and first order predicate logic
- We review the essentials of the *formal relational model* in this chapter
- In *practice*, there is a *standard model* based on SQL (Structured Query Language)
- Note: There are several important differences between the *formal* model and the *practical* model, as we shall see

# Relational Model Origins

- The model was first proposed by Dr. E.F. Codd of IBM Research in 1970 in the following paper:
  - "A Relational Model for Large Shared Data Banks," Communications of the ACM, June 1970
- The above paper caused a major revolution in the field of database management
- Dr. Codd earned the coveted ACM Turing Award in 1981

# Informal Definitions

- Informally, a **relation** looks like a **table** of values (see Figure 3.1 on next slide).
- A relation contains a **set of rows**.
- The data elements in each **row** represent certain facts that correspond to a real-world **entity** or **relationship**
  - In the formal model, rows are called **tuples**
- Each **column** has a column header that gives an indication of the meaning of the data items in that column
  - In the formal model, the column header is called an **attribute name** (or just **attribute**)



**Figure 3.1**

The attributes and tuples of a relation STUDENT.

# Informal Definitions

- Key of a Relation:
  - Each row (tuple) in the table is uniquely identified by the value of a particular attribute (or several attributes together)
    - Called the *key* of the relation
  - In the STUDENT relation, SSN is the key
  - If no attributes possess this uniqueness property, a new attribute can be added to the relation to assign unique row-id values (e.g. unique sequential numbers) to identify the rows in a relation
  - Called *artificial key* or *surrogate key*

# Formal Definitions – Relation Schema

- **Relation Schema** (or description) of a Relation:

- Denoted by  $R(A_1, A_2, \dots, A_n)$
  - $R$  is the **name** of the relation
  - The **attributes** of the relation are  $A_1, A_2, \dots, A_n$
  - $n$  is the **cardinality** of the relation

- Example:

CUSTOMER (Cust-id, Cust-name, Address, Phone#)

- CUSTOMER is the relation name
  - The CUSTOMER relation schema (or just *relation*) has four attributes: Cust-id, Cust-name, Address, Phone#
- Each attribute has a **domain** or a set of valid values.
  - For example, the domain of Cust-id can be 6 digit numbers.

# Formal Definitions - Tuple

- A **tuple** is an ordered set of values (enclosed in angled brackets ‘< ... >’)
- Each value is derived from an appropriate *domain*.
- A row in the CUSTOMER relation is a 4-tuple and would consist of four values, for example:
  - <632895, "John Smith", "101 Main St. Atlanta, GA 30332", "(404) 894-2000">
  - Called a 4-tuple because it has 4 values
  - In general, a particular relation will have n-tuples, where n is the number of attributes for the relation
- A relation is a **set** of such tuples (rows)

# Formal Definitions - Domain

- A **domain** of values can have a logical definition:
  - Example: “USA\_phone\_numbers” are the set of 10 digit phone numbers valid in the U.S.
- A domain also has a data-type or a format defined for it.
  - The USA\_phone\_numbers may have a format: (ddd)ddd-dddd where each d is a decimal digit.
  - Dates have various formats such as year, month, date formatted as yyyy-mm-dd, or as dd:mm:yyyy etc.
- The attribute name designates the **role** played by a domain in a relation:
  - Used to interpret the meaning of the data elements corresponding to that attribute
  - Example: The domain Date may be used to define two attributes “Invoice-date” and “Payment-date” with different meanings (roles)

# Formal Definitions – State of a Relation

- Formally, a **relation state**  $r(R)$  is a subset of the *Cartesian product* of the domains of its attributes
  - each domain contains the set of all possible values the attribute can take.
  - The *Cartesian product* contains all possible tuples from the attribute domains
  - The relations state  $r(R)$  is the subset of tuples that represent valid information in the mini-world at a particular time

# Formal Definitions - Summary

- Formally (see Figure 3.1),
  - Given relation schema  $R(A_1, A_2, \dots, A_n)$
  - Relation state  $r(R) \subset \text{dom}(A_1) \times \text{dom}(A_2) \times \dots \times \text{dom}(A_n)$
- $R(A_1, A_2, \dots, A_n)$  is the **schema** of the relation
- $R$  is the **name** of the relation
- $A_1, A_2, \dots, A_n$  are the **attributes** of the relation
- $r(R)$ : is a specific **state** (or "instance" or "population") of relation  $R$  – this is a *set of tuples* (rows) in the relation at a particular moment in time
  - $r(R) = \{t_1, t_2, \dots, t_n\}$  where each  $t_i$  is an  $n$ -tuple
  - $t_i = \langle v_1, v_2, \dots, v_n \rangle$  where each  $v_j$  element-of  $\text{dom}(A_j)$

# Formal Definitions - Example

- Let  $R(A_1, A_2)$  be a relation schema:
  - Let  $\text{dom}(A_1) = \{0,1\}$
  - Let  $\text{dom}(A_2) = \{a,b,c\}$
- Then: The *Cartesian product*  $\text{dom}(A_1) \times \text{dom}(A_2)$  contains all possible tuples from these domains:
$$\{\langle 0, a \rangle, \langle 0, b \rangle, \langle 0, c \rangle, \langle 1, a \rangle, \langle 1, b \rangle, \langle 1, c \rangle\}$$
- The relation state  $r(R) \subset \text{dom}(A_1) \times \text{dom}(A_2)$
- For example: One possible state  $r(R)$  could be  $\{\langle 0, a \rangle, \langle 0, b \rangle, \langle 1, c \rangle\}$ 
  - This state has three 2-tuples:  $\langle 0, a \rangle, \langle 0, b \rangle, \langle 1, c \rangle$

# Characteristics Of a Relation

- Ordering of tuples in a relation  $r(R)$ :
  - The tuples are *not considered to be ordered*, because a relation is a *set* of tuples (elements of a set are unordered) – see Figure 3.2
- Ordering of attributes in a relation schema  $R$  (and of values within each tuple):
  - We will consider the attributes in  $R(A_1, A_2, \dots, A_n)$  and the values in each  $t = \langle v_1, v_2, \dots, v_n \rangle$  to be **ordered**
  - However, a *more general definition* of relation (which we will not use) does not require attribute ordering
  - In this case, a tuple  $t = \{ \langle a_i, v_i \rangle, \dots, \langle a_j, v_j \rangle \}$  is an unordered set of  $n$   $\langle \text{attribute}, \text{value} \rangle$  pairs – one pair for each of the relation attributes

# Characteristics Of Relations (cont.)

- Values in a tuple:
  - All values are considered atomic (indivisible).
  - Each value must be from the domain of the attribute for that column
    - If tuple  $t = \langle v_1, v_2, \dots, v_n \rangle$  is a tuple (row) in the relation state  $r$  of  $R(A_1, A_2, \dots, A_n)$
    - Then each  $v_i$  must be a value from  $\text{dom}(A_i)$
  - A special **null** value is used to represent values that are unknown or inapplicable to certain tuples.

# Characteristics Of Relations (cont.)

- Notation:
  - We refer to **component values** of a tuple  $t$  by:
    - $t[A_i]$  or  $t.A_i$
    - This is the value  $v_i$  of attribute  $A_i$  for tuple  $t$
  - Similarly,  $t[A_u, A_v, \dots, A_w]$  refers to the subtuple of  $t$  containing the values of attributes  $A_u, A_v, \dots, A_w$ , respectively in  $t$

# Relational Integrity Constraints

- Constraints are **conditions** that must hold on **all valid** relation states.
- Constraints are *derived* from the mini-world semantics
- There are three *main types* of built-in constraints in the relational model:
  - **Key** constraints
  - **Entity integrity** constraints
  - **Referential integrity** constraints
- Another implicit constraint is the **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)

# Key Constraints

- **Superkey SK of R:**
  - Is a set of attributes SK of R with the following condition:
    - No two tuples in any valid relation state  $r(R)$  will have the same value for SK
    - That is, for any distinct tuples  $t_1$  and  $t_2$  in  $r(R)$ ,  $t_1.SK \neq t_2.SK$
    - This condition must hold in *any valid state*  $r(R)$
- **Key K of R:**
  - Is a "minimal" superkey
  - Formally, a key K is a superkey such that *removal of any attribute* from K results in a set of attributes that is not a superkey (or key) any more (does not possess the superkey uniqueness property)
  - Hence, a superkey *with one attribute* is always a key

# Key Constraints (cont.)

- Example: Consider the CAR relation schema:
  - CAR(State, Reg#, SerialNo, Make, Model, Year)
  - CAR has two keys (determined from the mini-world constraints):
    - Key1 = {State, Reg#}
    - Key2 = {SerialNo}
  - Both are also superkeys of CAR
  - However, {SerialNo, Make} is a superkey but *not* a key.
- In general:
  - Any *key* is a *superkey* (but not vice versa)
  - Any set of attributes that *includes a key* is a *superkey*
  - A *minimal* superkey is also a key

# Key Constraints (cont.)

- If a relation has several keys, they are called **candidate keys**; one is chosen to be the **primary key**; the others are called **unique** (or **secondary**) keys
  - The primary key attributes are underlined.
- Example: Consider the CAR relation schema:
  - CAR(State, Reg#, SerialNo, Make, Model, Year)
  - We choose License\_number (which contains (State, Reg#) together) as the primary key – see Figure 3.4
- The primary key value is used to *uniquely identify* each tuple in a relation
  - Provides the tuple *identity*
  - Also used to *reference* the tuple from other tuples
- General rule: Choose the smallest-sized candidate key (in bytes) as primary key
  - Not always applicable – choice is sometimes subjective

## CAR

<u>License_number</u>	Engine_serial_number	Make	Model	Year
Texas ABC-739	A69352	Ford	Mustang	02
Florida TVP-347	B43696	Oldsmobile	Cutlass	05
New York MPO-22	X83554	Oldsmobile	Delta	01
California 432-TFY	C43742	Mercedes	190-D	99
California RSK-629	Y82935	Toyota	Camry	04
Texas RSK-629	U028365	Jaguar	XJS	04

**Figure 3.4**

The CAR relation, with two candidate keys:  
License\_number and Engine\_serial\_number.

---

# Relational Database Schema

- **Relational Database Schema:**
  - A set S of relation schemas that belong to the same database.
  - S is the name of the whole **database schema**
  - $S = \{R_1, R_2, \dots, R_n\}$
  - $R_1, R_2, \dots, R_n$  are the names of the individual **relation schemas** within the database S
- Figure 3.5 shows a COMPANY database schema with 6 relation schemas

**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>	Dlocation
----------------	-----------

**PROJECT**

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

**WORKS\_ON**

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

**DEPENDENT**

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

**Figure 3.5**

Schema diagram for the COMPANY relational database schema.

---

# Relational Database State

- **Next two slides show an example of a COMPANY database state (Figure 3.6)**
  - Each relation has a set of tuples
    - The tuples in each table satisfy key and other constraints
    - If all constraints are satisfied by a database state, it is called a **valid state**
  - The database state changes to another state whenever the tuples in any relation are changed via insertions, deletions, or updates

### Figure 3.6

One possible database state for the COMPANY relational database schema.

#### EMPLOYEE

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

#### DEPARTMENT

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

#### DEPT\_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

### WORKS\_ON

<u>Essn</u>	<u>Pno</u>	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

### PROJECT

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

### DEPENDENT

<u>Essn</u>	<u>Dependent_name</u>	Sex	<u>Bdate</u>	<u>Relationship</u>
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

# Entity Integrity Constraint

- **Entity Integrity:**
  - The *primary key attributes* PK of each relation schema R in S cannot have null values in any tuple of  $r(R)$ .
    - This is because primary key values are used to *identify* the individual tuples.
    - $t.PK \neq \text{null}$  for any tuple  $t$  in  $r(R)$
    - If PK has several attributes, null is not allowed in any of these attributes
  - Note: Other attributes of R may be also be constrained to disallow null values (called NOT NULL constraint), even though they are not members of the primary key.

# Referential Integrity Constraint

- A constraint involving **two** relations
  - The previous constraints (key, entity integrity) involve a single relation.
- Used to specify a **relationship** among tuples in two relations:
  - The **referencing relation** and the **referenced relation**.

# Referential Integrity (cont.)

- 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.
  - A tuple t1 in R1 is said to **reference** a tuple t2 in R2 if  $t1.FK = t2.PK$
- Referential integrity can be displayed as a directed arc from R1.FK to R2.PK – see Figure 3.7

## 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>	Dependent_name	Sex	Bdate	Relationship
-------------	----------------	-----	-------	--------------

**Figure 3.7**

Referential integrity constraints displayed on the COMPANY relational database schema.

# Referential Integrity (cont.)

- Statement of the constraint
  - For a particular database state, the value of the foreign key attribute (or attributes) FK in each tuple of the **referencing relation** R1 can be **either**:
    - (1) An existing primary key (PK) value of a tuple in the **referenced relation** R2, or
    - (2) a **null**.
- In case (2), the FK in R1 should **not** be a part of its own primary key, and cannot have the NOT NULL constraint.

# Other Types of Constraints

- Semantic Integrity Constraints:
  - cannot be expressed by the built-in model constraints
  - Example: “the max. no. of hours per employee for all projects he or she works on is 56 hrs per week”
- A **constraint specification** language can be used to express these
- SQL has **TRIGGERS** and **ASSERTIONS** to express some of these constraints (see Section 5.2)

# Operations to Modify Relations

- Three basic operations:
  - INSERT
  - DELETE
  - UPDATE
- Integrity constraints should not be violated by the update operations.
- Several update operations may have to be grouped together into a **transaction**.
- Updates may **propagate** to cause other updates automatically. This may be necessary to maintain integrity constraints.

# Update Operations

- 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
  - Trigger additional updates so the violation is corrected
  - Execute a user-specified error-correction routine

# INSERT operation

- INSERT one or more new tuples into a relation
- INSERT may violate any of the constraints:
  - Domain constraint:
    - if one of the attribute values provided for a new tuple is not of the specified attribute domain
  - Key constraint:
    - if the value of a key attribute in a new tuple already exists in another tuple in the relation
  - Referential integrity:
    - if a foreign key value in a new tuple references a primary key value that does not exist in the referenced relation
  - Entity integrity:
    - if the primary key value is null in a new tuple

# DELETE operation

- DELETE one or more existing tuples from a relation
- DELETE may violate only referential integrity:
  - If the primary key value of the tuple being deleted is referenced from other tuples in the database
    - Can be remedied by several actions: RESTRICT, CASCADE, SET NULL
      - RESTRICT option: reject the deletion
      - CASCADE option: propagate the deletion by automatically deleting the referencing tuples
      - SET NULL option: set the foreign keys of the referencing tuples to NULL (the foreign keys cannot have NOT NULL constraint)

# DELETE operation

- DELETE one or more existing tuples from a relation
- DELETE may violate only referential integrity:
  - One of these options must be specified during database design for each referential integrity (foreign key) constraint (see below for example\*)

The screenshot shows a SQL Server Management Studio (SSMS) window with the title bar "SQLQuery1.sql - (loc...NCTIONX\pkatts (55))\* X Object Explorer Details". The main pane displays the following SQL script:

```
CREATE TABLE Persons
(
    PersonID int PRIMARY KEY NOT NULL,
    FirstName nvarchar(20),
    LastName nvarchar(20) NOT NULL,
    GenderID int FOREIGN KEY REFERENCES Genders(GenderID)
    ON DELETE SET NULL
);
GO
INSERT INTO Persons(PersonID, FirstName, LastName, GenderID)
VALUES(1, N'James', N'Palau', 1),
      (2, N'Ann', N'Nsang', 2),
      (3, N'Marc', N'Ulrich', 1),
      (4, N'Arjuh', N'Namdy', 3),
      (5, N'Aisha', N'Diabate', 2);
GO
SELECT ALL * FROM Persons;
GO
```

At the bottom of the window, there are tabs for "Results" and "Messages".

# UPDATE operation

- UPDATE modifies the values of attributes in one or more existing tuples in a relation
- UPDATE may violate domain constraint and NOT NULL constraint on an attribute being modified
- Other constraints may also be violated:
  - Updating the primary key (PK):
    - Similar to a DELETE followed by an INSERT
    - Need to specify similar options to DELETE
    - The CASCADE option propagates the new value of PK to the foreign keys of the referencing tuples automatically
  - Updating a foreign key (FK) may violate referential integrity
  - Updating an ordinary attribute (neither PK nor FK):
    - Can only violate domain or NOT NULL constraints

# Summary

- Presented Relational Model Concepts
  - Definitions
  - Characteristics of relations
- Discussed Relational Model Constraints and Relational Database Schemas
  - Domain constraints'
  - Key constraints
  - Entity integrity
  - Referential integrity
- Described the Relational Update Operations and Dealing with Constraint Violations

# Any Questions?

---



# Exercises

- Consider the figures on slide 80 and 81, discuss *all* integrity constraints violated by each of the following operations (if any) and the different ways to enforcing these constraints:

- Insert <‘Robert’, ‘F’, ‘Scott’, ‘943775543’, ‘1972-06-21’, ‘2365 Newcastle Rd, Bellaire, TX’, M, 58000, ‘888665555’, 1> into EMPLOYEE.
- Insert <‘ProductA’, 4, ‘Bellaire’, 2> into PROJECT.
- Insert <‘Production’, 4, ‘943775543’, ‘2007-10-01’> into DEPARTMENT.
- Insert <‘677678989’, NULL, ‘40.0’> into WORKS\_ON.
- Insert <‘453453453’, ‘John’, ‘M’, ‘1990-12-12’, ‘spouse’> into DEPENDENT.
- Delete the WORKS\_ON tuples with Essn = ‘333445555’.
- Delete the EMPLOYEE tuple with Ssn = ‘987654321’.
- Delete the PROJECT tuple with Pname = ‘ProductX’.
- Modify the Mgr\_ssn and Mgr\_start\_date of the DEPARTMENT tuple with Dnumber = 5 to ‘123456789’ and ‘2007-10-01’, respectively.
- Modify the Super\_ssn attribute of the EMPLOYEE tuple with Ssn = ‘999887777’ to ‘943775543’.
- Modify the Hours attribute of the WORKS\_ON tuple with Essn = ‘999887777’ and Pno = 10 to ‘5.0’.



# Part 2: Relational Databases

Kulsawasd Jitkajornwanich, PhD

kulsawasd.ji@kmitl.ac.th

[Taken and adapted from slides and/or other materials by Ramez Elmasri and Shamkant B. Navathe,  
“Fundamentals of Database Systems” (7<sup>th</sup> Edition), Addison-Wesley, Pearson]



# Chapter 3: Basic and Complex SQL



คณะวิทยาศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง  
Faculty of Science, King Mongkut's Institute of Technology Ladkrabang

# Outline

- SQL Data Definition (DDL: CREATE commands)
- Basic Database Modification (DML) in SQL
- Basic Retrieval Queries in SQL:
  - SELECT ... FROM ... WHERE ... statements
- Handling NULLs, 3-valued Logic in SQL
- Complex SQL:
  - Nested Queries (Correlated vs. uncorrelated; EXISTS)
  - Joined Tables, Inner Joins, and Outer Joins
  - Aggregate Functions and Grouping in SQL
- Summary of SQL Queries
- Additional Features of SQL
- Exercises

# SQL: Origins and History

- Evolution of the SQL Standard
  - First standard approved in 1989 (called SQL-89 or SQL1)
  - Comprehensive revision in 1992 (SQL-92 or SQL2)
  - Third big revision in 1999 (SQL-99, a trimmed-down version of a more ambitious revision known as SQL3)
  - Other revisions known as SQL:2003, SQL:2006 (XML added), SQL:2008 (OOP added), and SQL:2011.
- Origins of SQL
  - Originally called SEQUEL (Structured English Query Language), then SQL (Structured Query Language)
  - Developed at IBM Research for experimental relational DBMS called *System-R* in the 1970s

# SQL Fundamentals

- **SQL**
  - SQL is a standard, comprehensive language, based on the relational model
  - Considered one of the major reasons for the commercial success of relational databases
    - Originally called SEQUEL (Structured English Query Language), then SQL (Structured Query Language)
    - Developed at IBM Research for experimental relational DBMS called *System-R* in the 1970s
- SQL includes capabilities for many functions:
  - **DDL** statements for creating schemas and specifying data types and constraints
  - Statements for specifying database retrievals (**queries**)
  - Statements for modifying the database (**DML**)

# SQL Data Definition

- **CREATE** statement
  - Main SQL command for data definition
- CREATE (DDL) statement can be used to:
  - Create a named database schema
  - Create individual database tables and specify data types for the table attributes, as well as *key*, *referential integrity*, and *NOT NULL* constraints
  - Create named constraints
- Other commands can modify the tables and constraints
  - DROP and ALTER statements
    - CASCADE | RESTRICT (tables must be empty) options are also applied
    - For add'l commands (e.g., ADD COLUMN, DROP COLUMN, ALTER COLUMN, DROP CONSTRAINT), see Table 7.2 last two slides

# CREATE TABLE

- In its *simplest form*, specifies a new base relation by giving it a name, and specifying each of its attributes and their data types (INTEGER, FLOAT, DECIMAL(i,j), CHAR(n), VARCHAR(n), DATE, and other data types)
- A constraint NOT NULL may be specified on an attribute

```
CREATE TABLE DEPARTMENT (
    DNAME          VARCHAR(15)  NOT NULL,
    DNUMBER        INT          NOT NULL,
    MGRSSN         CHAR(9)      NOT NULL,
    MGRSTARTDATE   DATE        );
```

# CREATE TABLE (cont.)

- CREATE TABLE can also specify the primary key, UNIQUE keys, and referential integrity constraints (foreign keys)
- Via the PRIMARY KEY, UNIQUE, and FOREIGN KEY phrases

```
CREATE TABLE DEPARTMENT (
    DNAME          VARCHAR(15) NOT NULL,
    DNUMBER        INT         NOT NULL,
    MGRSSN         CHAR(9)     NOT NULL,
    MGRSTARTDATE   DATE,
    PRIMARY KEY (DNUMBER),
    UNIQUE (DNAME),
    FOREIGN KEY (MGRSSN) REFERENCES EMPLOYEE (SSN));
```

# Example: The COMPANY Database

- Figure in the next slide shows the COMPANY database *schema diagram*
- Referential integrity constraints shown as **directed edges** (--) from *foreign key* to *referenced relation (or parent table)*
- Primary key attributes of each table ***underlined***

## 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 3.7**

Referential integrity constraints displayed on the COMPANY relational database schema.

# Example: The COMPANY DDL

- Figures in following two slides shows *example DDL* for creating the tables of the COMPANY database
- Circular reference problem:
  - In the figure, some foreign keys cause *circular references*:
    - EMPLOYEE.Dno → DEPARTMENT.Dnumber
    - DEPARTMENT.Mgr\_ss → EMPLOYEE.Ssn
  - One of the tables must be created first without the FOREIGN KEY in the CREATE TABLE statement
  - The missing FOREIGN KEY can be added later using ALTER TABLE (as we will see later in the slides)

---

**CREATE TABLE EMPLOYEE**

( Fname	VARCHAR(15)	NOT NULL,
Minit	CHAR,	
Lname	VARCHAR(15)	NOT NULL,
Ssn	CHAR(9)	NOT NULL,
Bdate	DATE,	
Address	VARCHAR(30),	
Sex	CHAR,	
Salary	DECIMAL(10,2),	
Super_ssn	CHAR(9),	
Dno	INT	NOT NULL,

**PRIMARY KEY** (Ssn),

**FOREIGN KEY** (Super\_ssn) **REFERENCES** EMPLOYEE(Ssn),

**FOREIGN KEY** (Dno) **REFERENCES** DEPARTMENT(Dnumber) );

**CREATE TABLE DEPARTMENT**

( Dname	VARCHAR(15)	NOT NULL,
Dnumber	INT	NOT NULL,
Mgr_ssn	CHAR(9)	NOT NULL,
Mgr_start_date	DATE,	

**PRIMARY KEY** (Dnumber),

**UNIQUE** (Dname),

**FOREIGN KEY** (Mgr\_ssn) **REFERENCES** EMPLOYEE(Ssn) );

**CREATE TABLE DEPT\_LOCATIONS**

( Dnumber	INT	NOT NULL,
Dlocation	VARCHAR(15)	NOT NULL,

**PRIMARY KEY** (Dnumber, Dlocation),

**FOREIGN KEY** (Dnumber) **REFERENCES** DEPARTMENT(Dnumber) );

**Figure 4.1**

SQL CREATE TABLE data definition statements for defining the COMPANY schema from Figure 3.7.

**Continued next page...**

## **CREATE TABLE PROJECT**

( Pname	VARCHAR(15)	<b>NOT NULL,</b>
Pnumber	INT	<b>NOT NULL,</b>
Plocation	VARCHAR(15),	
Dnum	INT	<b>NOT NULL,</b>

**PRIMARY KEY** (Pnumber),

**UNIQUE** (Pname),

**FOREIGN KEY** (Dnum) **REFERENCES** DEPARTMENT(Dnumber) );

## **CREATE TABLE WORKS\_ON**

( Essn	CHAR(9)	<b>NOT NULL,</b>
Pno	INT	<b>NOT NULL,</b>
Hours	DECIMAL(3,1)	<b>NOT NULL,</b>

**PRIMARY KEY** (Essn, Pno),

**FOREIGN KEY** (Essn) **REFERENCES** EMPLOYEE(Ssn),

**FOREIGN KEY** (Pno) **REFERENCES** PROJECT(Pnumber) );

## **CREATE TABLE DEPENDENT**

( Essn	CHAR(9)	<b>NOT NULL,</b>
Dependent_name	VARCHAR(15)	<b>NOT NULL,</b>
Sex	CHAR,	
Bdate	DATE,	
Relationship	VARCHAR(8),	

**PRIMARY KEY** (Essn, Dependent\_name),

**FOREIGN KEY** (Essn) **REFERENCES** EMPLOYEE(Ssn) );

# The CREATE SCHEMA Statement

- A DBMS can manage multiple databases
- A database can have several schemas, each of which serves a single database application.
- Any two tables from different schemas can have the same name.

**CREATE SCHEMA COMPANY AUTHORIZATION 'Smith' ;**

- User 'Smith' will be “owner” of COMPANY schema, can create tables, other constructs in that schema
- Table names can be prefixed by schema name if multiple schemas exist (e.g. COMPANY.EMPLOYEE)
- Prefix not needed if there is a “default schema” for the user

# Basic SQL Data Types

- Basic numeric data types:
  - Integers: INTEGER (or INT), SMALLINT
  - Real (floating point): FLOAT (or REAL), DOUBLE PRECISION
  - Formatted: DECIMAL(i,j) (or DEC (i,j) or NUMERIC(i,j)) specify i total decimal digits, j after decimal point
    - i called *precision*, j called *scale*
- Basic character string data types:
  - Fixed-length: CHAR(n) or CHARACTER(n)
  - Variable-length: VARCHAR(n) or CHAR VARYING(n) or CHARACTER VARYING(n)

# SQL Data Types (cont)

- DATE data type:
  - Standard DATE formatted as yyyy-mm-dd
  - For example, DATE '2010-01-22'
  - Older formats still used in some systems, such as 'JAN-22-2010'
  - Values are ordered, with later dates larger than earlier ones
- TIME data type:
  - Standard TIME formatted as hh:mm:ss
  - E.g., TIME '15:20:22' represents 3.20pm and 22 seconds
  - TIME(i) includes an additional i-1 decimal digits for fractions of a second
  - E.g., TIME(5) value could be '15:20:22.1234'
- TIMESTAMP data type:
  - A DATE combined with a TIME(i), where the default i=7
  - For example, TIMESTAMP '2010-01-22 15:20:22.123456'
  - A different i>7 can be specified if needed

# Specifying Referential Integrity Options and Naming of Constraints

- We can specify RESTRICT (the default), CASCADE, SET NULL or SET DEFAULT on referential integrity constraints (foreign keys)
- Separate options for ON DELETE, ON UPDATE
  - Figure in the next slide gives some examples
- A constraint can be given a *constraint name*; this allows to DROP the constraint later
  - Figure in the next slide also illustrates naming of constraints

```

CREATE TABLE EMPLOYEE
(
    ...,
    Dno      INT          NOT NULL      DEFAULT 1,
    CONSTRAINT EMPPK
        PRIMARY KEY (Ssn),
    CONSTRAINT EMPSUPERFK
        FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn)
            ON DELETE SET NULL      ON UPDATE CASCADE,
    CONSTRAINT EMPDEPTFK
        FOREIGN KEY(Dno) REFERENCES DEPARTMENT(Dnumber)
            ON DELETE SET DEFAULT   ON UPDATE CASCADE);
CREATE TABLE DEPARTMENT
(
    ...,
    Mgr_ssn  CHAR(9)      NOT NULL      DEFAULT '888665555',
    ...,
    CONSTRAINT DEPTPK
        PRIMARY KEY(Dnumber),
    CONSTRAINT DEPTSK
        UNIQUE (Dname),
    CONSTRAINT DEPTMGRFK
        FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn)
            ON DELETE SET DEFAULT  ON UPDATE CASCADE);
CREATE TABLE DEPT_LOCATIONS
(
    ...,
    PRIMARY KEY (Dnumber, Dlocation),
    FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber)
        ON DELETE CASCADE      ON UPDATE CASCADE);

```

**Figure 4.2**  
 Example illustrating  
 how default attribute  
 values and referential  
 integrity triggered  
 actions are specified  
 in SQL.

# Specifying DOMAIN Constraints Using the CHECK Clause

- Another type of constraint can be specified using CHECK
  - E.g., `CHECK (Dnumber > 0 AND Dnumber < 21)` can be specified after the `Dnumber` attribute
  - Alternatively, a special domain can be created with a domain name using `CREATE DOMAIN`
    - E.g. `CREATE DOMAIN D_NUM AS INTEGER CHECK (Dnumber > 0 AND Dnumber < 21);`
    - `D_NUM` can now be used as the data type for the `Dnumber` attribute of the `DEPARTMENT` table, as well as for `Dnum` of `PROJECT`, `Dno` of `EMPLOYEE`, and so on
    - `CHECK` can also be used to specify more general constraints

# Basic Retrieval Queries in SQL

- SQL basic statement for retrieving information from a database is the **SELECT** statement
- Important distinction between *practical SQL model* and the *formal relational model*:
  - SQL allows a table (relation) to have two or more tuples that are identical in all their attribute values
  - Hence, an SQL relation (table) is a **multi-set** (sometimes called a **bag**) of tuples; it is *not* a set of tuples
- SQL relations can be *constrained to be sets* by specifying PRIMARY KEY or UNIQUE attributes, or by using the DISTINCT option in a query

# Bag (Multiset) versus Set

- A **bag** or **multi-set** is like a set, but an element may appear more than once
  - Example:  $\{A, B, C, A\}$  is a bag (but *not* a set).  $\{A, B, C\}$  is a bag (and also a set).
  - Bags also resemble lists, but the order is irrelevant in a bag.
- Example:
  - $\{A, B, A\} = \{B, A, A\}$  as bags
  - However,  $[A, B, A]$  is not equal to  $[B, A, A]$  as lists

# Basic Retrieval Queries in SQL (cont.)

- Simplest form of the SQL SELECT statement is called a *mapping* or a **SELECT-FROM-WHERE block**
- Our examples use the COMPANY database schema
- **SELECT**      <attribute list>  
**FROM**        <table list>  
**WHERE**      <condition>
  - <attribute list> is a list of attribute names whose values are to be retrieved by the query
  - <table list> is a list of the table (relation) names required to process the query
  - <condition> is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query

# Simple SQL Queries (cont.)

- Query text ends with a semi-colon
- First example of a simple query on one table (relation)
- Query 0: Retrieve the birthdate and address of the employee whose name is 'John B. Smith'
- Use the EMPLOYEE table only

```
Q0:  SELECT      BDATE, ADDRESS  
      FROM        EMPLOYEE  
      WHERE       FNAME='John' AND MINIT='B'  
                  AND LNAME='Smith' ;
```

**Figure 3.6**

One possible database state for the COMPANY relational database schema.

**EMPLOYEE**

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

**DEPARTMENT**

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

**DEPT\_LOCATIONS**

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

**Continued next page...**

### WORKS\_ON

<u>Essn</u>	<u>Pno</u>	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

### PROJECT

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

### DEPENDENT

<u>Essn</u>	<u>Dependent_name</u>	<u>Sex</u>	<u>Bdate</u>	<u>Relationship</u>
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

# Simple SQL Queries (cont.)

- Example of a query that uses two tables
- Query 1: Retrieve the name and address of all employees who work for the 'Research' department.

```
Q1: SELECT      FNAME, LNAME, ADDRESS  
        FROM       EMPLOYEE, DEPARTMENT  
        WHERE      DNAME='Research' AND DNUMBER=DNO ;
```

- (DNAME='Research') is called a *selection condition*
- (DNUMBER=DNO) is called a *join condition* (it *joins* two tuples from EMPLOYEE and DEPARTMENT tables whenever EMPLOYEE.DNO=DEPARTMENT.DNUMBER)

# Simple SQL Queries (cont.)

- A *selection condition* refers to attributes from a single table, and selects (chooses) those records that satisfy the condition
- A *join condition generally* refers to attributes from two tables, and joins (or combines) pairs of records that satisfy the condition
- In the previous query:
  - (DNAME='Research') chooses the DEPARTMENT record
  - (DNUMBER=DNO) joins the record with each EMPLOYEE record that works for that department

# Simple SQL Queries (cont.)

- Query 2 (needs 3 tables): For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birthdate.

Q2: SELECT PNUMBER, DNUM, LNAME, BDATE, ADDRESS  
FROM PROJECT, DEPARTMENT, EMPLOYEE  
WHERE DNUM=DNUMBER AND MGRSSN=SSN  
AND PLOCATION='Stafford' ;

- In Q2, there are two join conditions
- The join condition DNUM=DNUMBER relates a PROJECT record to its controlling DEPARTMENT record
- The join condition MGRSSN=SSN relates the controlling DEPARTMENT to the EMPLOYEE who manages that department

# Aliasing Table Names Using Tuple Variables

- An *alias* (or *tuple variable*) can be used instead of the table name when prefixing attribute names
- Example:

Query Q1 can be rewritten as follows using the aliases D for DEPARTMENT and E for EMPLOYEE:

```
SELECT      E.FNAME, E.LNAME, E.ADDRESS  
FROM        EMPLOYEE AS E, DEPARTMENT AS D  
WHERE       D.DNAME='Research' AND D.DNUMBER=E.DNO ;
```

# Missing WHERE-clause

- The *WHERE-clause* is **optional** in an SQL query
- A *missing WHERE-clause* indicates no condition; hence, all tuples of the relations in the FROM-clause are selected
  - This is equivalent to the condition WHERE TRUE
- Example: Retrieve the SSN values for all employees.

```
Q9: SELECT      SSN  
          FROM       EMPLOYEE ;
```

- If more than one relation is specified in the FROM-clause *and* there is no WHERE-clause (hence no join conditions), then *all possible combinations* of tuples are joined together (known as *CARTESIAN PRODUCT* of the relations)

# Retrieving all the Attributes Using Asterisk (\*)

- To retrieve all the attribute values of the selected tuples, a \* (asterisk) is used, which stands for *all the attributes*

Examples:

```
Q1C: SELECT      *
      FROM        EMPLOYEE
      WHERE       DNO=5 ;
```

```
Q1D: SELECT      *
      FROM        EMPLOYEE, DEPARTMENT
      WHERE       DNAME='Research' AND
                  DNO=DNUMBER ;
```

# Eliminating Duplicates Using DISTINCT

- As mentioned earlier, SQL does not treat a relation as a set but a multiset (or bag); duplicate tuples can appear
- To eliminate duplicate tuples in a query result, the keyword **DISTINCT** is used
- Example: Result of Q11 may have duplicate SALARY values; result of Q11A does not have any duplicate values

Q11:     SELECT     SALARY  
              FROM     EMPLOYEE

Q11A:    SELECT    **DISTINCT** SALARY  
            FROM    EMPLOYEE

# Set and Multiset Operations in SQL

- SQL has directly incorporated some set operations
- The set operations are: union (UNION), set difference (EXCEPT or MINUS) and intersection (INTERSECT)
- *Note that: the set operands will be converted to set before performing the operation*
- Results of these set operations are sets of tuples; *duplicate tuples are eliminated from the result*
- Set operations apply only to *type compatible* relations (also called *union compatible*); the two relations must have the same attribute types and in the same order
- Set operations typically applied to the results of two separate queries; e.g Q1 UNION Q2

# Set Operations (cont.)

- Example: Query 4: Make a list of all project names for projects that involve an employee whose last name is 'Smith' as a worker on the project or as a manager of the department that controls the project.

```
Q4:   (SELECT      PNAME
        FROM       PROJECT, DEPARTMENT, EMPLOYEE
        WHERE      DNUM=DNUMBER AND
                  MGR_SSN=SSN AND LNAME='Wong')
        UNION
        (SELECT      PNAME
        FROM       PROJECT, WORKS_ON, EMPLOYEE
        WHERE      PNUMBER=PNO AND
                  ESSN=SSN AND LNAME='Wong') ;
```

# Multiset Operations in SQL

- SQL has multiset operations when the user does not want to eliminate duplicates from the query results
- These are: UNION ALL, EXCEPT ALL, and INTERSECT ALL; see examples in Figure 4.5 (next slide)
- Results of these operations are multisets of tuples; *all tuples and duplicates in the input tables are considered* when computing the result
- Multiset operations also apply only to *type compatible* relations
- Typically applied to the results of two separate queries; e.g Q1  
UNION ALL Q2 ;

(a) Two tables, R(A) and S(A), are shown. Both have column A.

R
A
a1
a2
a2
a3

S
A
a1
a2
a4
a5

(b) The result of  $R(A) \cup S(A)$  is table T.

T
A
a1
a1
a2
a2
a2
a3
a4
a5

(c) The result of  $R(A) - S(A)$  is table T.

T
A
a2
a3

(d) The result of  $R(A) \cap S(A)$  is table T.

T
A
a1
a2

**Figure 4.5**

The results of SQL multiset operations. (a) Two tables, R(A) and S(A). (b)  $R(A) \cup S(A)$ . (c)  $R(A) - S(A)$ . (d)  $R(A) \cap S(A)$ .

# Substring Comparison Conditions

- The **LIKE** comparison operator is used to compare partial strings
- Two reserved characters are used: '\*' (or '%' in some implementations) replaces an arbitrary number of characters, and '\_' replaces a single arbitrary character
- Conditions can be used in WHERE-clause

# Substring Comparison Conditions (cont.)

- Example: Query 25: Retrieve all employees whose address is in Houston, Texas. Here, the value of the ADDRESS attribute must contain the substring 'Houston, TX' in it.

```
Q25:   SELECT  FNAME, LNAME
          FROM    EMPLOYEE
          WHERE   ADDRESS LIKE '*Houston, TX*' ;
```

# Substring Comparison Conditions (cont.)

- Example: Query 26: Retrieve all employees who were born during the 1950s.
  - Here, '5' must be the 3<sup>rd</sup> character of the string (according to the standard format for DATE yyyy-mm-dd), so the BDATE value is '\_ \_ 5 \_ \_ \_ \_ \_', with each underscore as a place holder for a single arbitrary character.

```
Q26:   SELECT      FNAME, LNAME
          FROM       EMPLOYEE
          WHERE      BDATE LIKE'_ _ _ _ _ 5 _';
```

- The LIKE operator allows users to get around the fact that each value is considered atomic and indivisible
  - Hence, in SQL, character string attribute values are not atomic

# Applying Arithmetic in SQL Queries

- The standard arithmetic operators '+', '-'. '\*' , and '/' (for addition, subtraction, multiplication, and division, respectively) can be applied to numeric attributes and values in an SQL query
- Example: Query 27: Show the effect of giving all employees who work on the 'ProductX' project a 10% raise.

```
Q27:    SELECT      FNAME, LNAME, 1.1*SALARY  
            FROM       EMPLOYEE, WORKS_ON, PROJECT  
            WHERE      SSN=ESSN AND PNO=PNUMBER  
                      AND PNAME='ProductX' ;
```

# Ordering a query result using the ORDER BY clause

- The **ORDER BY** clause is used to sort the tuples in a query result based on the values of some attribute(s)
- Example: Query 28: Retrieve a list of employees and the projects each works in, ordered by the employee's department, and within each department ordered alphabetically by employee last name.

```
Q28:    SELECT      DNAME, LNAME, FNAME, PNAME  
            FROM        DEPARTMENT, EMPLOYEE,  
                        WORKS_ON, PROJECT  
            WHERE       DNUMBER=DNO AND SSN=ESSN  
                        AND PNO=PNUMBER  
            ORDER BY   DNAME, LNAME DESC;
```

# ORDER BY (cont.)

- The default order is in ascending order of values
- We can specify the keyword **DESC** if we want a descending order; the keyword **ASC** can be used to explicitly specify ascending order, even though it is the default
- Without ORDER BY, the rows in a query result appear in some system-determined (random) order

# SQL Statements for Updating the Database

- There are three SQL commands to modify the database:  
**INSERT, DELETE, and UPDATE**
- INSERT is used for adding one or more records to a table
- DELETE is for removing records
- UPDATE is for modifying existing records
- Some operations may be rejected if they violate integrity constraints; others may propagate additional updating automatically if specified in the database schema

# INSERT statement (cont.)

- Values for the attributes should be listed in the same order as the attributes were specified in the **CREATE TABLE** command
- Attributes that have default values can be omitted in the new record(s)
- Example:

```
U1: INSERT INTO EMPLOYEE  
      VALUES ('Richard','K','Marini', '653298653', '30-DEC-52',  
      '98 Oak Forest,Katy,TX', 'M', 37000,'987654321', 4 );
```

- An alternate form of INSERT specifies explicitly the attribute names that correspond to the values in the new tuple
  - Attributes with NULL or default values can be left out
- Example: Insert a tuple for a new EMPLOYEE for whom we only know the FNAME, LNAME, and SSN attributes.

```
U1A: INSERT INTO EMPLOYEE (FNAME, LNAME, SSN)  
      VALUES ('Richard', 'Marini', '653298653');
```

# INSERT statement (cont.)

- Can also insert tuples from a query result into a table
  - Example: Suppose we want to create a temporary table that has the employee last name, project name, and hours per week for each employee.
    - A table WORKS\_ON\_INFO is created by U3A, and is loaded with the summary information retrieved from the database by the query in U3B.

```
U3A: CREATE TABLE WORKS_ON_INFO  
        (EMP_NAME          VARCHAR(15),  
         PROJ_NAME         VARCHAR(15),  
         HOURS_PER_WEEK   DECIMAL(3,1));
```

U3B:    INSERT INTO WORKS\_ON\_INFO (EMP\_NAME, PROJ\_NAME,  
                        HOURS\_PER\_WEEK)  
SELECT E.LNAME, P.PNAME, W.HOURS  
FROM EMPLOYEE E, PROJECT P, WORKS\_ON W  
WHERE E.SSN=W.ESSN AND W.PNO=P.PNUMBER ;

# DELETE Statement

- Removes tuples from a relation
  - Includes a WHERE-clause to select the tuples to be deleted
  - Referential integrity should be enforced (via REJECT, CASCADE, SET NULL, or SET DEFAULT)
  - Tuples are deleted from only *one table* at a time (unless CASCADE is specified on a referential integrity constraint)
  - Missing WHERE-clause deletes *all tuples* in the relation; the table then becomes an empty table
  - Number of tuples deleted is the number of tuples selected by the WHERE-clause

# DELETE Statement (cont.)

- Examples:

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 Statement

- Used to modify attribute values of one or more selected tuples
- A WHERE-clause selects the tuples to be modified
- An additional SET-clause specifies the attributes to be modified and their new values
- Each command modifies tuples *in the same relation*
- Referential integrity should be enforced (via REJECT, CASCADE, SET NULL, or SET DEFAULT)

# UPDATE Statement (cont.)

- Example: Change the location and controlling department number of project number 10 to 'Bellaire' and 5, respectively.

```
U5: UPDATE PROJECT  
      SET      PLOCATION = 'Bellaire',  
              DNUM = 5  
      WHERE    PNUMBER=10 ;
```

# UPDATE Statement (cont.)

- Example: Give all employees in department number 5 a 10% raise in salary

```
U6: UPDATE    EMPLOYEE  
        SET          SALARY = SALARY *1.1  
        WHERE        DNO = 5 ;
```

- In this request, the modified SALARY value depends on the original SALARY value in each tuple
  - The reference to the SALARY attribute on the right of = refers to the old SALARY value before modification
  - The reference to the SALARY attribute on the left of = refers to the new SALARY value after modification

# Handling NULLs in SQL

- SQL allows queries that check if an attribute is **NULL** (missing or undefined or not applicable)
- SQL uses **IS** or **IS NOT** to compare an attribute to NULL because it considers each NULL value distinct from other NULL values, so *equality comparison is not appropriate.*
- Example: Query 14: Retrieve the names of all employees who do not have supervisors.

```
Q14:    SELECT      FNAME, LNAME  
            FROM        EMPLOYEE  
            WHERE       SUPERSSN IS NULL ;
```

# 3-valued Logic in SQL

- Standard 2-valued logic assumes a condition can evaluate to either TRUE or FALSE
- With NULLs a condition can evaluate to UNKNOWN, leading to 3-valued logic
- Combining individual conditions using AND, OR, NOT logical connectives must consider UNKNOWN in addition to TRUE and FALSE
- Example: Consider a condition EMPLOYEE.DNO = 5; this evaluates for individual tuples in EMPLOYEE as follows:
  - TRUE for tuples with DNO=5
  - UNKNOWN for tuples where DNO is NULL
  - FALSE for other tuples in EMPLOYEE

**Table 5.1** Logical Connectives in Three-Valued Logic

(a)	AND	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	FALSE	UNKNOWN
	FALSE	FALSE	FALSE	FALSE
	UNKNOWN	UNKNOWN	FALSE	UNKNOWN

(b)	OR	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	TRUE	TRUE
	FALSE	TRUE	FALSE	UNKNOWN
	UNKNOWN	TRUE	UNKNOWN	UNKNOWN

(c)	NOT			
	TRUE	FALSE		
	FALSE	TRUE		
	UNKNOWN	UNKNOWN		

# Nesting of Queries in SQL

- A complete SELECT ... query, called a *nested query*, can be specified within the WHERE-clause of *another query*
  - The other query is called the *outer query*
  - Many of the previous queries can be specified in an alternative form using nesting
- Query 1: Retrieve the name and address of all employees who work for the 'Research' department.

```
Q1:SELECT      FNAME, LNAME, ADDRESS  
              FROM  
              WHERE      DNO IN      (SELECT      DNUMBER  
                                FROM  
                                WHERE      DNAME='Research' ) ;  
                                DEPARTMENT
```

# Nesting of Queries (cont.)

- In Q1, the *nested query* selects the DNUMBER of the 'Research' department
- The *outer query* select an EMPLOYEE tuple if its DNO value is in the result of the nested query
- The comparison operator **IN** compares a value v with a set (or multi-set) of values V, and evaluates to TRUE if v is one of the elements in V
- In general, can have several levels of nested queries
- A reference to an *unqualified attribute* refers to the relation declared in the *innermost nested query*
- In this example, the nested query is *not correlated* with the outer query
- **NOTE:** In Oracle, uncorrelated query performance can be further improved by incorporating WITH-Clause to create temporary tables.

# Correlated Nested Queries

- If a condition in the WHERE-clause of a *nested query* references an attribute of a relation declared in the *outer query*, the two queries are said to be *correlated*
  - The result of a correlated nested query is *different* for each tuple (or combination of tuples) of the relation(s) the outer query
- Query 12: Retrieve the name of each employee who has a dependent with the same first name as the employee.

```
Q12: 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) ;
```

# Correlated Nested Queries (cont.)

- In Q12, the nested query has a different result *for each tuple* in the outer query (because it refers to E.FNAME)
- A query written with nested SELECT... FROM... WHERE... blocks and using the = or IN comparison operators can *always* be expressed as a single block query.
- For example, Q12 may be written as in Q12A

```
Q12A:      SELECT      E.FNAME, E.LNAME  
              FROM        EMPLOYEE E, DEPENDENT D  
              WHERE       E.SSN=D.ESSN AND  
                         E.FNAME=D.DEPENDENT_NAME ;
```

# Correlated Nested Queries (cont.)

- The original SQL as specified for SYSTEM R also had a **CONTAINS** comparison operator, which is used in conjunction with nested correlated queries
  - This operator was *dropped from the language*, possibly because of the difficulty in implementing it efficiently
  - Most implementations of SQL *do not have* this operator
  - The CONTAINS operator compares *two sets of values*, and returns TRUE if one set contains all values in the other set

# Correlated Nested Queries (cont.)

- Example of Using CONTAINS (not in current SQL)
- Query 3: Retrieve the name of each employee who works on *all the projects* controlled by department number 5.

```
Q3:   SELECT      E.FNAME, E.LNAME
          FROM        EMPLOYEE AS E
          WHERE       (SELECT      W.PNO
                        FROM        WORKS_ON AS W
                        WHERE       E.SSN=W.ESSN)
                      CONTAINS
                      (SELECT      P.PNUMBER
                        FROM        PROJECT AS P
                        WHERE       P.DNUM=5)    ) ;
```

# Correlated Nested Queries (cont.)

- In Q3, the second nested query, which is *not correlated* with the outer query, retrieves the project numbers of all projects controlled by department 5
- The first nested query, which is *correlated*, retrieves the project numbers on which the employee works; this is *different for each employee tuple* because it references E.SSN

# The EXISTS Function in SQL

- EXISTS is used to check whether the result of a query is empty (contains no tuples) or not (contains one or more tuples)
  - Applied to a query, but returns a boolean result (TRUE or FALSE)
  - Can be used in the WHERE-clause as a condition
  - EXISTS (Q) evaluates to TRUE if the result of Q has one or more tuple; evaluates to FALSE if the result of Q has no tuples

# The EXISTS Function (cont.)

- Query 7: Retrieve the names of employees who are department managers *and* have at least one dependent.

```
Q7: SELECT      M.FNAME, M.LNAME
      FROM        EMPLOYEE AS M
      WHERE       EXISTS (SELECT    *
                           FROM        DEPENDENT
                           WHERE       M.SSN=ESSN)
      AND EXISTS (SELECT    *
                           FROM        DEPARTMENT
                           WHERE       M.SSN=MGRSSN) ;
```

# The EXISTS Function (cont.)

- Query 6: Retrieve the names of employees who have no dependents.

```
Q6:   SELECT      E.FNAME, E.LNAME  
        FROM       EMPLOYEE E  
        WHERE      NOT EXISTS  (SELECT      *  
                                FROM       DEPENDENT D  
                                WHERE      E.SSN=D.ESSN) ;
```

- In Q6, the correlated nested query retrieves all DEPENDENT tuples related to an EMPLOYEE tuple. If *none exist*, the EMPLOYEE tuple is selected
  - EXISTS is necessary for the expressive power of SQL

# Explicit (Literal) Sets in SQL

- An **explicit (enumerated) set of values** is enclosed in parentheses
- Query 13: Retrieve the social security numbers of all employees who work on project number 1, 2, or 3.

```
Q13:SELECT      DISTINCT ESSN  
                  FROM      WORKS_ON  
                  WHERE      PNO IN (1, 2, 3);
```

# Joined Tables (Relations) in SQL

- Can specify a "joined relation" in the FROM-clause
  - Looks like any other relation but is the result of a join
  - Allows the user to specify different types of joins (INNER JOIN, NATURAL JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, CROSS JOIN, etc) – see the next slides
  - Each join type can specify a different query and produce a different result

# Types of join – INNER JOIN

- This is the regular join operation
- Joined tuples must satisfy all join conditions
- Example: Query QJ1: Retrieve the employee names with the names of the department they work for

```
SELECT      E.FNAME, E.LNAME, D.DNAME  
FROM        DEPARTMENT AS D, EMPLOYEE AS E  
WHERE      D.DNUMBER=E.DNO ;
```

- This can be written using *joined tables* as follows:

```
SELECT      E.FNAME, E.LNAME, D.DNAME  
FROM        (DEPARTMENT AS D JOIN  
                      EMPLOYEE AS E ON D.DNUMBER=E.DNO) ;
```

# Types of join – OUTER JOIN

- In QJ1, an EMPLOYEE record is joined only if it has a matching DEPARTMENT with D.DNUMBER=E.DNO
- Hence, an EMPLOYEE with NULL for E.DNO will not appear in the query result
- Also, a DEPARTMENT that has no matching EMPLOYEE records (i.e. currently has no employees) does not appear in the query result
- OUTER JOINs gives the options to include every EMPLOYEE record or every DEPARTMENT record in the query results
- A record that does not have a matching joined record will be “padded” with an imaginary “NULL record” from the other table (all its attributes will be NULL)

# Types of join – LEFT OUTER JOIN

- Example: Query QJ2: Retrieve the employee names with the names of the department they work for; *every department* must appear in the result even if it has no employees

This can be written using *joined tables* as follows:

```
SELECT      E.FNAME, E.LNAME, D.DNAME  
FROM        (DEPARTMENT AS D LEFT OUTER JOIN  
             EMPLOYEE AS E ON D.DNUMBER=E.DNO) ;
```

- Note: An earlier left outer join syntax in ORACLE is as follows:

```
SELECT      E.FNAME, E.LNAME, D.DNAME  
FROM        DEPARTMENT AS D, EMPLOYEE AS E  
WHERE      D.DNUMBER +
```

# Types of join – RIGHT OUTER JOIN

- Example: Query QJ3: Retrieve the employee names with the names of the department they work for; *every employee* must appear in the result even they are not currently assigned to a department

This can be written using *joined tables* as follows:

```
SELECT      E.FNAME, E.LNAME, D.DNAME  
FROM        (DEPARTMENT AS D RIGHT OUTER JOIN  
                  EMPLOYEE AS E ON D.DNUMBER=E.DNO) ;
```

Note: An earlier left outer join syntax in ORACLE is as follows:

```
SELECT      E.FNAME, E.LNAME, D.DNAME  
FROM        DEPARTMENT AS D, EMPLOYEE AS E  
WHERE      D.DNUMBER =+ E.DNO ;
```

# Types of join – FULL OUTER JOIN

- Example: Query QJ4: Retrieve the employee names with the names of the department they work for; *every employee and every department must appear in the result*

This can be written using *joined tables* as follows:

```
SELECT      E.FNAME, E.LNAME, D.DNAME  
FROM        (DEPARTMENT AS D FULL OUTER JOIN  
             EMPLOYEE AS E ON D.DNUMBER=E.DNO) ;
```

- Note: An earlier left outer join syntax in ORACLE is as follows:

```
SELECT      E.FNAME, E.LNAME, D.DNAME  
FROM        DEPARTMENT AS D, EMPLOYEE AS E  
WHERE      D.DNUMBER + = + E.DNO ;
```

# Types of join – NATURAL JOIN

- If the join attributes in both tables *have the same name*, the join condition can be left out (it is automatically added by the system)
- NATURAL JOIN is a form of inner join
- Example: QJ5: We rename DNUMBER in DEPARTMENT to DNO to match the join attribute name (DNO) in EMPLOYEE (we also rename other attributes)
- Implicit join condition is E.DNO = D.DNO

```
SELECT      E.FN, E.LN, E.ADR  
FROM        (DEPARTMENT AS D(DNM, DNO, MSSN, STRDATE)  
                    NATURAL JOIN EMPLOYEE AS  
                    E(FN,MI,LN,S,BD,ADR,SX,SAL,SU,DNO) ;
```

# Joined Tables – Other Examples

- Query 8: Retrieve the employee names, and the names of their direct supervisor

```
Q8: SELECT E.FNAME, E.LNAME, S.FNAME, S.LNAME  
      FROM (EMPLOYEE AS E INNER JOIN EMPLOYEE AS S  
            ON E.SUPERSSN=S.SSN);
```

- In Q8, an *employee with no supervisor* will not appear in the result; if we want every employee to appear, we write:

```
Q8':SELECT E.FNAME, E.LNAME, S.FNAME, S.LNAME  
      FROM (EMPLOYEE E LEFT OUTER JOIN  
            EMPLOYEE S ON E.SUPERSSN=S.SSN)
```

# Joined Tables – Other Examples

Examples:

Q1: SELECT FNAME, LNAME, ADDRESS  
FROM EMPLOYEE, DEPARTMENT  
WHERE DNAME='Research' AND DNUMBER=DNO ;

- could be written as:

Q1: SELECT FNAME, LNAME, ADDRESS  
FROM (EMPLOYEE JOIN DEPARTMENT  
ON DNUMBER=DNO)  
WHERE DNAME='Research' ;

- or as:

Q1: SELECT FNAME, LNAME, ADDRESS  
FROM (EMPLOYEE NATURAL JOIN DEPARTMENT  
AS DEPT(DNAME, DNO, MSSN, MSDATE))  
WHERE DNAME='Research' ;

# Joined Tables – Other Examples

- Another Example: Q2 could be written as follows; this illustrates multiple joins in the joined tables

```
Q2: SELECT      PNUMBER, DNUM, LNAME, BDATE, ADDRESS  
      FROM        ((PROJECT JOIN DEPARTMENT ON  
                  DNUM=DNUMBER)  
                      JOIN EMPLOYEE ON MGRSSN=SSN) )  
      WHERE       PLOCATION='Stafford' ;
```

# Aggregate Functions

- Include **COUNT, SUM, MAX, MIN, and AVG**
- These can summarize information from multiple tuples into a single tuple
- Query 15: Find the maximum salary, the minimum salary, and the average salary among all employees.

```
Q15:    SELECT    MAX(SALARY) AS HIGH_SAL,  
                  MIN(SALARY) AS LOW_SAL,  
                  AVG(SALARY) AS MEAN_SAL  
            FROM      EMPLOYEE ;
```

# Aggregate Functions (cont.)

- Query 16: Find the maximum salary, the minimum salary, and the average salary among employees who work for the 'Research' department.

```
Q16:   SELECT  MAX(E.SALARY), MIN(E.SALARY),
              AVG(E.SALARY)
        FROM    EMPLOYEE E, DEPARTMENT D
        WHERE   E.DNO=D.DNUMBER AND
              D.DNAME='Research' ;
```

# Aggregate Functions (cont.)

- Queries 17 and 18: Retrieve the total number of employees in the company (Q17), and the number of employees in the 'Research' department (Q18). (Note: COUNT(\*) counts the number of selected records)

Q17:      SELECT      COUNT (\*)  
                FROM      EMPLOYEE ;

Q18:      SELECT      COUNT (\*)  
                FROM      EMPLOYEE AS E, DEPARTMENT AS D  
                WHERE     E.DNO=D.DNUMBER AND  
                      D.DNAME='Research' ;

# Grouping (Partitioning Records into Subgroups)

- In many cases, we want to apply the aggregate functions to *subgroups of tuples* in a relation
- Each subgroup of tuples consists of the set of tuples that have the *same value* for the *grouping attribute(s)* – for example, *employees who work in the same department* (have the same DNO)
- The aggregate functions are applied to each subgroup independently
- SQL has a **GROUP BY**-clause for specifying the grouping attributes, which *must also appear in the SELECT-clause*

# Grouping (cont.)

- Query 20: For each department, retrieve the department number, the number of employees in the department, and their average salary.

```
Q20:      SELECT      DNO, COUNT (*), AVG (SALARY)
              FROM        EMPLOYEE
              GROUP BY    DNO ;
```

- In Q20, the EMPLOYEE tuples are divided into groups
  - Each group has same value for the grouping attribute DNO
- The COUNT and AVG functions are applied to each such group of tuples separately (see next slide)
- The SELECT-clause includes only the grouping attribute and the functions to be applied on each group of tuples

**Figure 5.1**

Results of GROUP BY and HAVING. (a) Q24. (b) Q26.

(a)

Fname	Minit	Lname	Ssn	...	Salary	Super_ssn	Dno	
John	B	Smith	123456789	...	30000	333445555	5	
Franklin	T	Wong	333445555		40000	888665555	5	
Ramesh	K	Narayan	666884444		38000	333445555	5	
Joyce	A	English	453453453		25000	333445555	5	
Alicia	J	Zelaya	999887777		25000	987654321	4	
Jennifer	S	Wallace	987654321		43000	888665555	4	
Ahmad	V	Jabbar	987987987		25000	987654321	4	
James	E	Bong	888665555		55000	NULL	1	

Grouping EMPLOYEE tuples by the value of Dno

Dno	Count (*)	Avg (Salary)
5	4	33250
4	3	31000
1	1	55000

Result of Q24

(b)

Pname	Pnumber	...	Essn	Pno	Hours	
ProductX	1	...	123456789	1	32.5	
ProductX	1		453453453	1	20.0	
ProductY	2		123456789	2	7.5	
ProductY	2		453453453	2	20.0	
ProductY	2		333445555	2	10.0	
ProductZ	3		666884444	3	40.0	
ProductZ	3		333445555	3	10.0	
Computerization	10		333445555	10	10.0	
Computerization	10		999887777	10	10.0	
Computerization	10		987987987	10	35.0	
Reorganization	20		333445555	20	10.0	
Reorganization	20		987654321	20	15.0	
Reorganization	20		888665555	20	NULL	
Newbenefits	30		987987987	30	5.0	
Newbenefits	30		987654321	30	20.0	
Newbenefits	30		999887777	30	30.0	

These groups are not selected by the HAVING condition of Q26.

**Continued next page...**

After applying the WHERE clause but before applying HAVING

# Grouping (cont.)

- A join condition can be used with grouping
- Query 21: For each project, retrieve the project number, project name, and the number of employees who work on that project.

```
Q21:    SELECT      P.PNUMBER, P.PNAME, COUNT (*)
          FROM        PROJECT AS P,WORKS_ON AS W
          WHERE       P.PNUMBER=W.PNO
          GROUP BY   P.PNUMBER, P.PNAME ;
```

- In this case, the grouping and aggregate functions are applied *after* the joining of the two relations

# The HAVING-Clause

- Sometimes we want to retrieve the values of these aggregate functions for only those *groups that satisfy certain conditions*
- The **HAVING**-clause is used for specifying a selection condition on groups (rather than on individual tuples)
- Query 22: For each project *on which more than two employees work*, retrieve the project number, project name, and the number of employees who work on that project (Figure 5.1(b) – next two slides).

```
Q22:      SELECT          PNUMBER, PNAME, COUNT(*)  
              FROM           PROJECT, WORKS_ON  
              WHERE          PNUMBER=PNO  
              GROUP BY       PNUMBER, PNAME  
              HAVING         COUNT(*) > 2 ;
```

The diagram illustrates a query execution plan. On the left is a source table with columns Pname, Pnumber, ..., Essn, Pno, and Hours. On the right is a result table with columns Pname and Count (\*). Arrows show the mapping of rows from the source table to the result table based on the Pname column.

**Source Table:**

Pname	Pnumber	...	Essn	Pno	Hours
ProductY	2	...	123456789	2	7.5
ProductY	2	...	453453453	2	20.0
ProductY	2	...	333445555	2	10.0
Computerization	10	...	333445555	10	10.0
Computerization	10	...	999887777	10	10.0
Computerization	10	...	987987987	10	35.0
Reorganization	20	...	333445555	20	10.0
Reorganization	20	...	987654321	20	15.0
Reorganization	20	...	888665555	20	NULL
Newbenefits	30	...	987987987	30	5.0
Newbenefits	30	...	987654321	30	20.0
Newbenefits	30	...	999887777	30	30.0

**Result Table:**

Pname	Count (*)
ProductY	3
Computerization	3
Reorganization	3
Newbenefits	3

**Text:** Result of Q26  
(Pnumber not shown)

After applying the HAVING clause condition

# Summary of SQL Commands

- A query in SQL can consist of up to six clauses, but only the first two, SELECT and FROM, are mandatory. The clauses are specified in the following order:

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

# Summary of SQL Commands (cont.)

- The SELECT-clause lists the attributes or functions to be retrieved
- The FROM-clause specifies all relations (or aliases) needed in the query but not those needed in nested queries, as well as joined tables
- The WHERE-clause specifies the conditions for selection and join of tuples from the relations specified in the FROM-clause
- GROUP BY specifies grouping attributes
- HAVING specifies a condition for selection of groups
- ORDER BY specifies an order for displaying the query result
  - Conceptually, a query is evaluated by first applying the WHERE-clause, then GROUP BY and HAVING, and finally the SELECT-clause and ORDER BY

# Summary of SQL Commands (cont.)

**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 Queries (cont.)

**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.

# Additional Features of SQL

- Techniques for specifying complex retrieval queries (see Ch.7)
- Writing programs in various programming languages that include SQL statements: Embedded and dynamic SQL, SQL/CLI (Call Level Interface) and its predecessor ODBC, SQL/PSM (Persistent Stored Module) (See Ch.10)
- Set of commands for specifying physical database design parameters, file structures for relations, and access paths, e.g., CREATE INDEX (Ch.16, 17)
- Transaction control commands (Ch.20)
- Specifying the granting and revoking of privileges to users (Ch.30)
- Constructs for creating triggers (Ch.5,26)
- Enhanced relational systems known as object-relational define relations as classes. Abstract data types (called User Defined Types- UDTs) are supported with CREATE TYPE (Ch.12)
- New technologies such as XML (Ch.13) and OLAP (Ch.29) are added to versions of SQL

# Any Questions?

---



# Exercises

- 1) Discuss the various reasons that lead to the occurrence of NULL values in relations.
- 2) Discuss how NULLs are treated in comparison operators in SQL. How are NULLs treated when aggregate functions are applied in an SQL query? How are NULLs treated if they exist in grouping attributes?
- 3) Consider a database state on slides 24 - 25 and write SQLs to:
  - a) Retrieve the names of all employees in department 5 who work more than 10 hours per week on the ProductX project.
  - b) List the names of all employees who have a dependent with the same first name as themselves.
  - c) Find the names of all employees who are directly supervised by ‘Franklin Wong’.
  - d) Insert your information as a new employee



# Part 2: Relational Databases

Kulsawasd Jitkajornwanich, PhD

kulsawasd.ji@kmitl.ac.th

[Taken and adapted from slides and/or other materials by Ramez Elmasri and Shamkant B. Navathe,  
“Fundamentals of Database Systems” (7<sup>th</sup> Edition), Addison-Wesley, Pearson]



# Chapter 4: Conceptual Database Design Using ER and EER

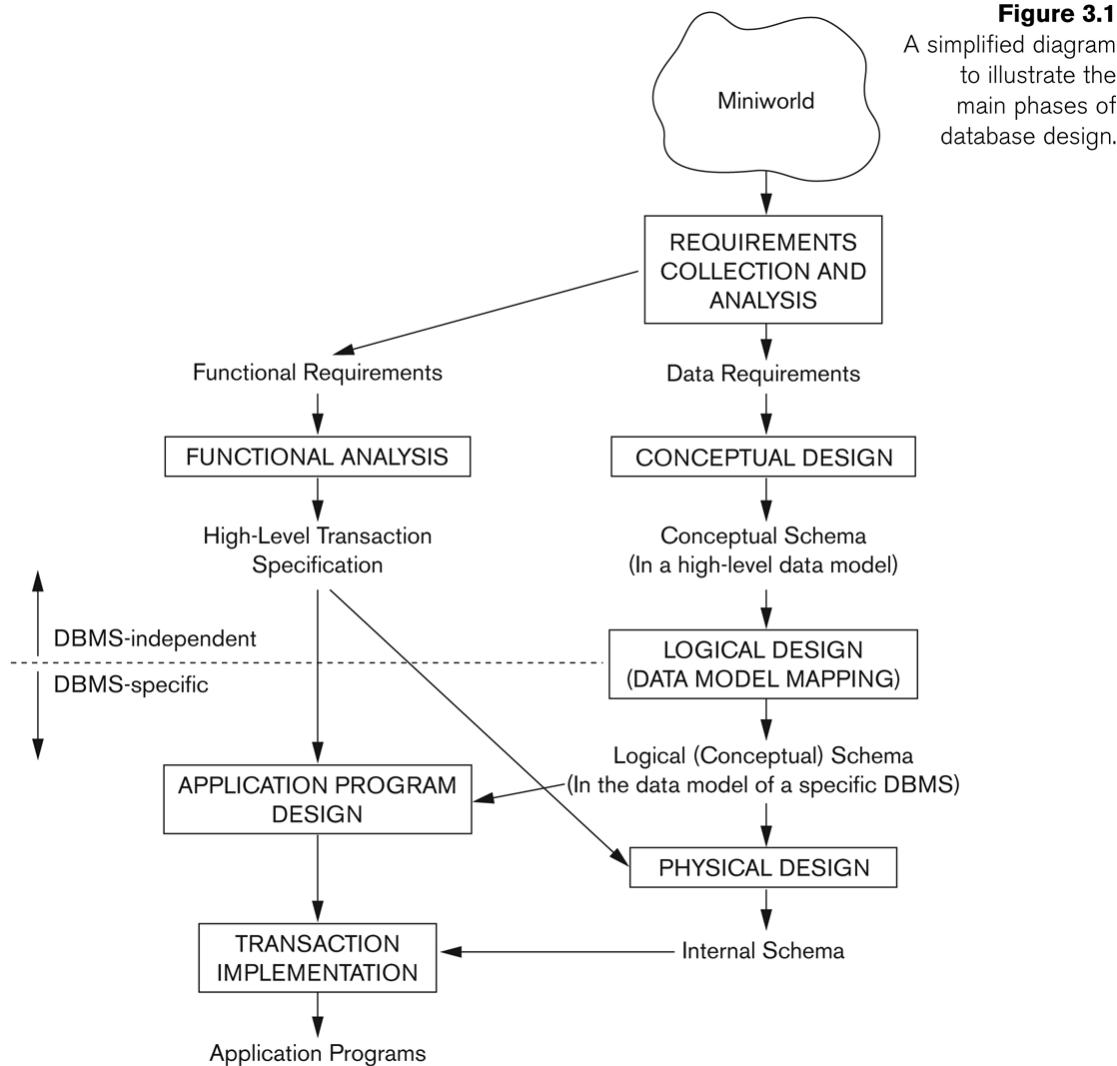


คณะวิทยาศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง  
Faculty of Science, King Mongkut's Institute of Technology Ladkrabang

# Outline

- Overview of Database Design Process
- ER Model Concepts
  - Entities and Attributes
  - Relationships and Relationship Types
  - Weak Entity Types
- ER Diagrams – Notation
- EER Model Concepts
  - subclasses/superclasses
  - specialization/generalization
  - categories (UNION types)
- Constraints on Specialization/Generalization

# Overview of Database Design Process



**Figure 3.1**

A simplified diagram to illustrate the main phases of database design.

# Example COMPANY Database

- We need to create a database schema design based on the following (simplified) **requirements** of the COMPANY Database:
  - The company is organized into DEPARTMENTS. Each department has a name, number and an employee who *manages* the department. We keep track of the start date of the department manager. A department may have several locations.
  - Each department *controls* a number of PROJECTS. Each project has a unique name, unique number and is located at a single location.

# Example COMPANY Database (Continued)

- The database will store each EMPLOYEE's social security number, address, salary, sex, and birthdate.
  - Each employee *works for* one department but may *work on* several projects.
  - The DB will keep track of the number of hours per week that an employee currently works on each project.
  - It is required to keep track of the *direct supervisor* of each employee.
- Each employee may *have* a number of DEPENDENTS.
  - For each dependent, the DB keeps a record of name, sex, birthdate, and relationship to the employee.

# ER Model Concepts

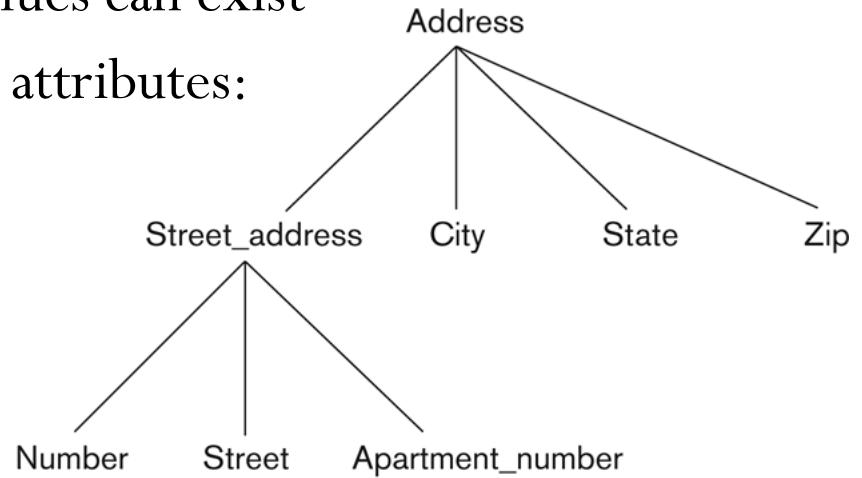
- Entities and Attributes
  - Entity is a basic concept for the ER model. Entities are specific things or objects in the mini-world that are represented in the database.
    - For example the EMPLOYEE John Smith, the Research DEPARTMENT, the ProductX PROJECT
  - Attributes are properties used to describe an entity.
    - For example an EMPLOYEE entity may have the attributes Name, SSN, Address, Sex, BirthDate
  - A specific entity will have a value for each of its attributes.
    - For example a specific employee entity may have Name='John Smith', SSN='123456789', Address ='731, Fondren, Houston, TX', Sex='M', BirthDate='09-JAN-55'
  - Each attribute has a *value set* (or data type) associated with it – e.g. integer, string, date, enumerated type, ...

# Types of Attributes (1)

- Simple
  - Each entity has a single atomic value for the attribute. For example, SSN or Sex.
- Composite
  - The attribute may be composed of several components. For example:
    - Address(Apt#, House#, Street, City, State, ZipCode, Country), or
    - Name(FirstName, MiddleName, LastName).
    - 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} .

# Types of Attributes (2)

- In general, composite and multi-valued attributes may be nested arbitrarily to any number of levels, although this is rare.
  - For example, PreviousDegrees of a STUDENT is a composite multi-valued attribute denoted by {PreviousDegrees (College, Year, Degree, Field)}
  - Multiple PreviousDegrees values can exist
  - Each has four subcomponent attributes:
    - College, Year, Degree, Field



# Entity Types and Key Attributes (1)

- Entities with the same basic attributes are grouped or typed into an entity type.
  - For example, the entity type EMPLOYEE and PROJECT.
- An attribute of an entity type for which each entity must have a unique value is called a key attribute of the entity type.
  - For example, SSN of EMPLOYEE.

# Entity Types and Key Attributes (2)

- A key attribute may be composite.
  - VehicleTagNumber is a key of the CAR entity type with components (Number, State).
- An entity type may have more than one key.
  - The CAR entity type may have two keys:
    - VehicleIdentificationNumber (popularly called VIN)
    - VehicleTagNumber (Number, State), aka license plate number.
- Each key is underlined (Note: this is different from the relational schema where only one “primary key” is underlined).

# Entity Set

- Each entity type will have a collection of entities stored in the database
  - Called the **entity set** or sometimes **entity collection**
- Previous slide shows three CAR entity instances in the entity set for CAR
- Same name (CAR) used to refer to both the entity type and the entity set
- However, entity type and entity set may be given different names
- Entity set is the current *state* of the entities of that type that are stored in the database

# Value Sets (Domains) of Attributes

- Each simple attribute is associated with a value set
  - E.g., Lastname has a value which is a character string of upto 15 characters
  - Date has a value consisting of MM-DD-YYYY where each letter is an integer
- A **value set** specifies the set of values associated with an attribute

# Attributes and Value Sets

- Value sets are similar to data types in most programming languages – e.g., integer, character (n), real, bit
- Mathematically, an attribute A for an entity type E whose value set is V is defined as a function

$$A : E \rightarrow P(V)$$

Where  $P(V)$  indicates a power set (which means all possible subsets) of V. The above definition covers simple and multivalued attributes.

- We refer to the value of attribute A for entity e as  $A(e)$ .

# Displaying an Entity type

- In ER diagrams, an entity type is displayed in a rectangular box
- Attributes are displayed in ovals
  - Each attribute is connected to its entity type
  - Components of a composite attribute are connected to the oval representing the composite attribute
  - Each key attribute is underlined
  - Multivalued attributes displayed in double ovals
- See the full ER notation in advance on the next slide

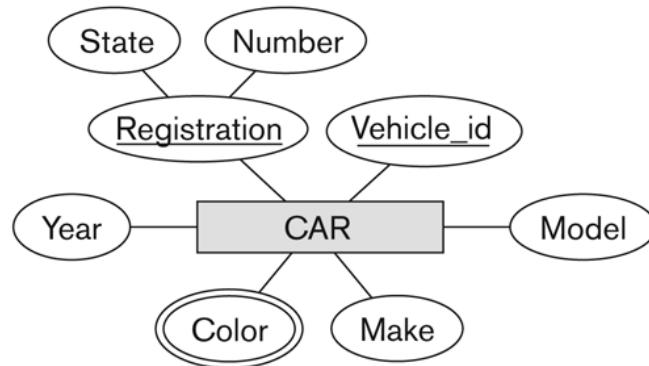
# NOTATION for ER diagrams

**Figure 3.14**  
Summary of the notation for ER diagrams.

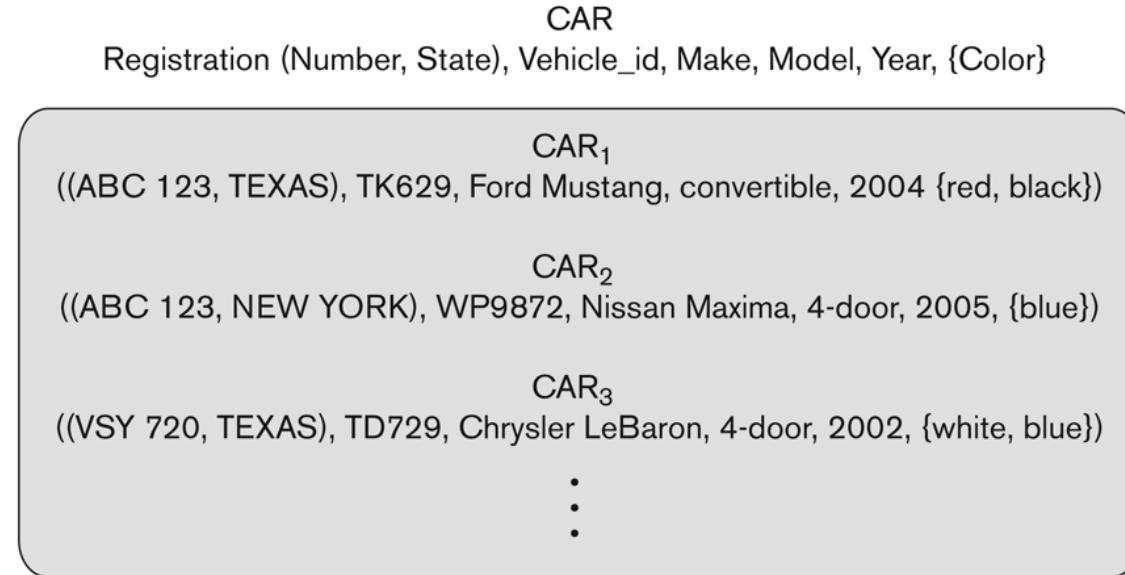
Symbol	Meaning
	Entity
	Weak Entity
	Relationship
	Identifying Relationship
	Attribute
	Key Attribute
	Multivalued Attribute
	Composite Attribute
	Derived Attribute
	Total Participation of $E_2$ in $R$
	Cardinality Ratio 1: N for $E_1:E_2$ in $R$
	Structural Constraint (min, max) on Participation of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

# Entity Type CAR with two keys and a corresponding Entity Set

(a)



(b)



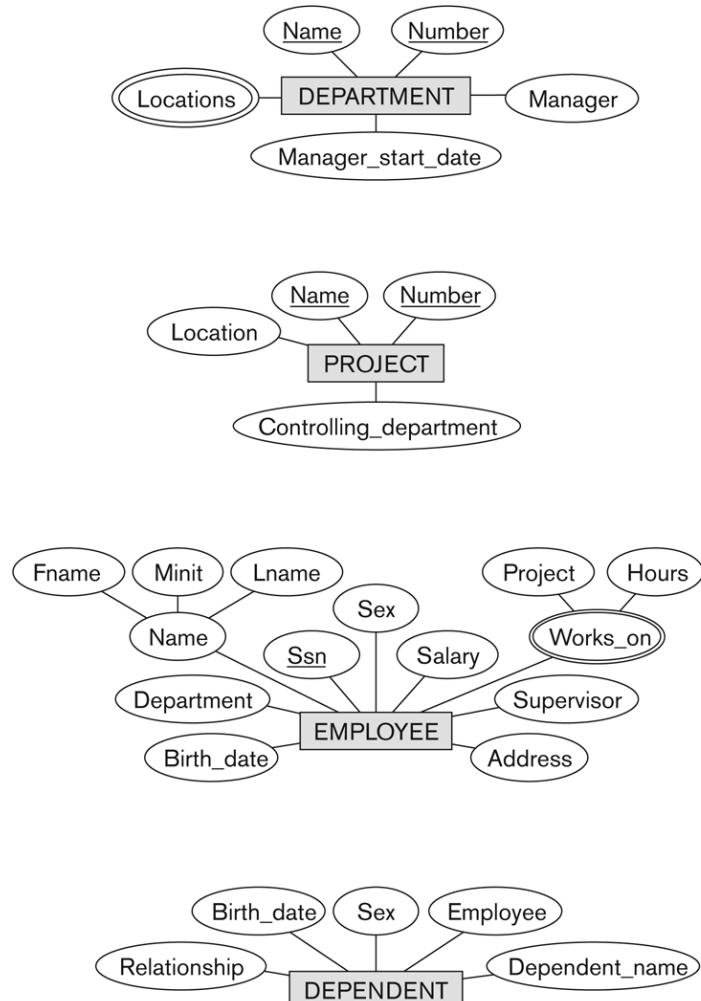
**Figure 3.7**

The CAR entity type with two key attributes, Registration and Vehicle\_id. (a) ER diagram notation. (b) Entity set with three entities.

# Initial Conceptual Design of Entity Types for the COMPANY Database Schema

- Based on the requirements, we can identify four initial entity types in the COMPANY database:
  - DEPARTMENT
  - PROJECT
  - EMPLOYEE
  - DEPENDENT
- Their initial conceptual design is shown on the following slide
- The initial attributes shown are derived from the requirements description

# Initial Design of Entity Types: EMPLOYEE, DEPARTMENT, PROJECT, DEPENDENT



**Figure 3.8**

Preliminary design of entity types for the COMPANY database. Some of the shown attributes will be refined into relationships.

# Refining the initial design by introducing relationships

- The initial design is typically not complete
- Some aspects in the requirements will be represented as **relationships**
- ER model has three main concepts:
  - Entities (and their entity types and entity sets)
  - Attributes (simple, composite, multivalued)
  - Relationships (and their relationship types and relationship sets)
- We introduce relationship concepts next

# Relationships and Relationship Types (1)

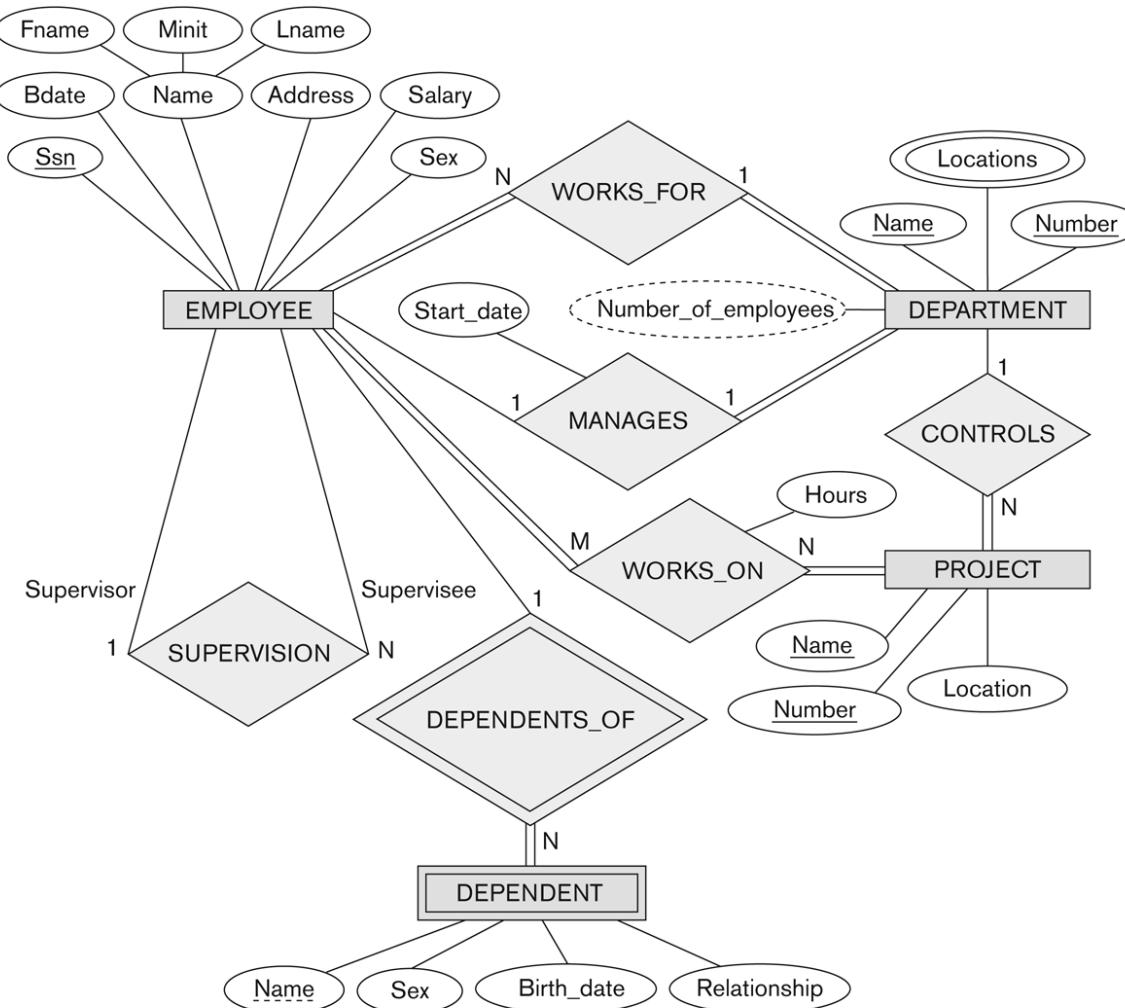
- A **relationship** relates two or more distinct entities with a specific meaning.
  - For example, EMPLOYEE John Smith *works on* the ProductX PROJECT, or EMPLOYEE Franklin Wong *manages* the Research DEPARTMENT.
- Relationships of the same type are grouped or typed into a **relationship type**.
  - For example, the WORKS\_ON relationship type in which EMPLOYEES and PROJECTs participate, or the MANAGES relationship type in which EMPLOYEES and DEPARTMENTS participate.
- The degree of a relationship type is the number of participating entity types.
  - Both MANAGES and WORKS\_ON are *binary* relationships.

# Refining the COMPANY database schema by introducing relationships

- By examining the requirements, six relationship types are identified
- All are *binary* relationships( degree 2)
- Listed below with their participating entity types:
  - WORKS\_FOR (between EMPLOYEE, DEPARTMENT)
  - MANAGES (also between EMPLOYEE, DEPARTMENT)
  - CONTROLS (between DEPARTMENT, PROJECT)
  - WORKS\_ON (between EMPLOYEE, PROJECT)
  - SUPERVISION (between EMPLOYEE (as subordinate), EMPLOYEE (as supervisor))
  - DEPENDENTS\_OF (between EMPLOYEE, DEPENDENT)

# ER DIAGRAM – Relationship Types are:

WORKS\_FOR, MANAGES, WORKS\_ON, CONTROLS, SUPERVISION, DEPENDENTS\_OF



# Discussion on Relationship Types

- In the refined design, some attributes from the initial entity types are refined into relationships:
  - Manager of DEPARTMENT -> MANAGES
  - Works\_on of EMPLOYEE -> WORKS\_ON
  - Department of EMPLOYEE -> WORKS\_FOR
  - etc
- In general, more than one relationship type can exist between the same participating entity types
  - MANAGES and WORKS\_FOR are distinct relationship types between EMPLOYEE and DEPARTMENT
  - Different meanings and different relationship instances.

# Constraints on Relationships

- Constraints on Relationship Types
  - (Also known as ratio constraints)
  - Cardinality Ratio (specifies *maximum* participation)
    - One-to-one (1:1)
    - One-to-many (1:N) or Many-to-one (N:1)
    - Many-to-many (M:N)
- Existence Dependency Constraint (specifies *minimum* participation)  
(also called participation constraint)
  - zero (optional participation, not existence-dependent)
  - one or more (mandatory participation, existence-dependent)

# Recursive Relationship Type

- A relationship type between the same participating entity type in **distinct roles**
- Also called a **self-referencing** relationship type.
- Example: the SUPERVISION relationship
- EMPLOYEE participates twice in two distinct roles:
  - supervisor (or boss) role
  - supervisee (or subordinate) role
- Each relationship instance relates two distinct EMPLOYEE entities:
  - One employee in *supervisor* role
  - One employee in *supervisee* role

# Displaying a recursive relationship

- In a recursive relationship type.
  - Both participations are same entity type in different roles.
  - For example, SUPERVISION relationships between EMPLOYEE (in role of supervisor or boss) and (another) EMPLOYEE (in role of subordinate or worker).
- In ER diagram, need to display role names to distinguish participations.

# Weak Entity Types

- An entity that does not have a key attribute and that 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
- **Example:**
  - A DEPENDENT entity is identified by the dependent's first name, *and* the specific EMPLOYEE with whom the dependent is related
  - Name of DEPENDENT is the *partial key*
  - DEPENDENT is a *weak entity type*
  - EMPLOYEE is its identifying entity type via the identifying relationship type DEPENDENT\_OF

# Attributes of Relationship types

- A relationship type can have attributes:
  - For example, HoursPerWeek of WORKS\_ON
  - Its value for each relationship instance describes the number of hours per week that an EMPLOYEE works on a PROJECT.
    - A value of HoursPerWeek depends on a particular (employee, project) combination
- Most relationship attributes are used with M:N relationships
  - In 1:N relationships, they can be transferred to the entity type on the N-side of the relationship

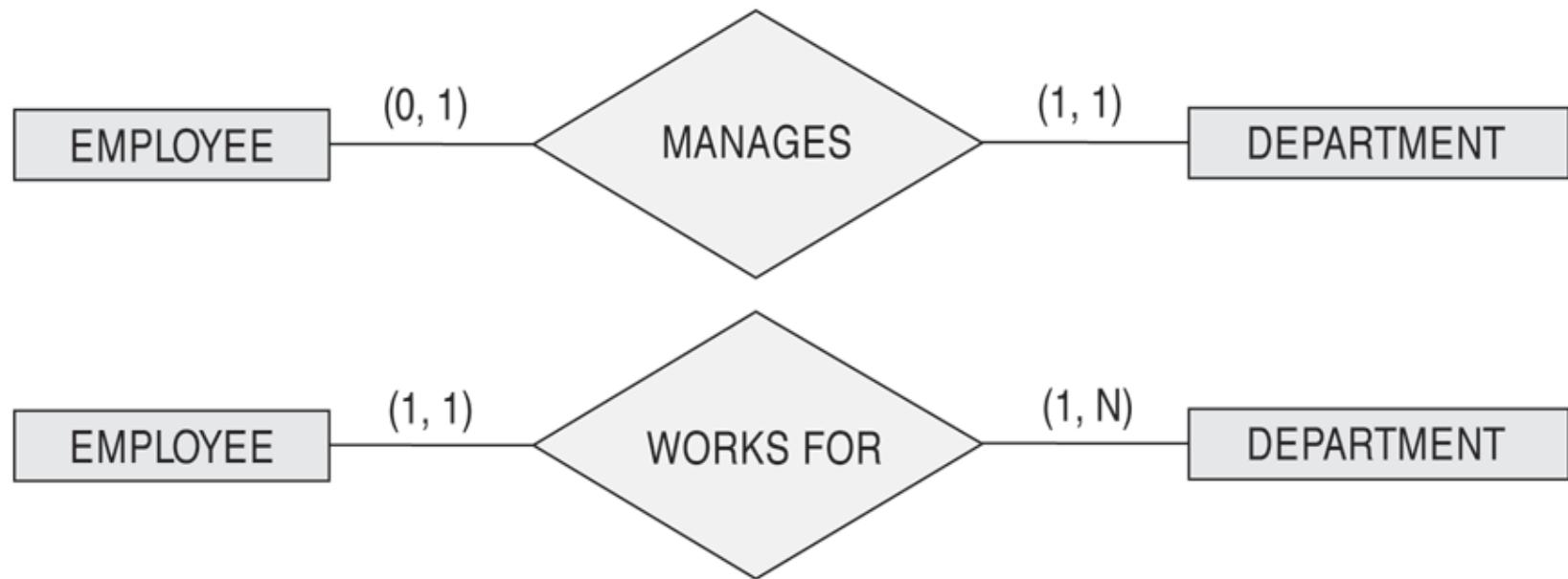
# Notation for Constraints on Relationships

- Cardinality ratio (of a binary relationship): 1:1, 1:N, N:1, or M:N
  - Shown by placing appropriate numbers on the relationship edges.
- Participation constraint (on each participating entity type): total (called existence dependency) or partial.
  - Total shown by double line, partial by single line.
- NOTE: These are easy to specify for Binary Relationship Types.

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

- Specified on each participation of an entity type E in a relationship type R
- Specifies that each entity e in E participates in at least *min* and at most *max* relationship instances in R
- Default(no constraint):  $\text{min}=0$ ,  $\text{max}=n$  (signifying no limit)
- Must have  $\text{min} \leq \text{max}$ ,  $\text{min} \geq 0$ ,  $\text{max} \geq 1$
- Derived from the knowledge of mini-world 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 of employees.
    - Specify (1,1) for participation of EMPLOYEE in WORKS\_FOR
    - Specify (0,n) for participation of DEPARTMENT in WORKS\_FOR

# The (min,max) notation for relationship constraints

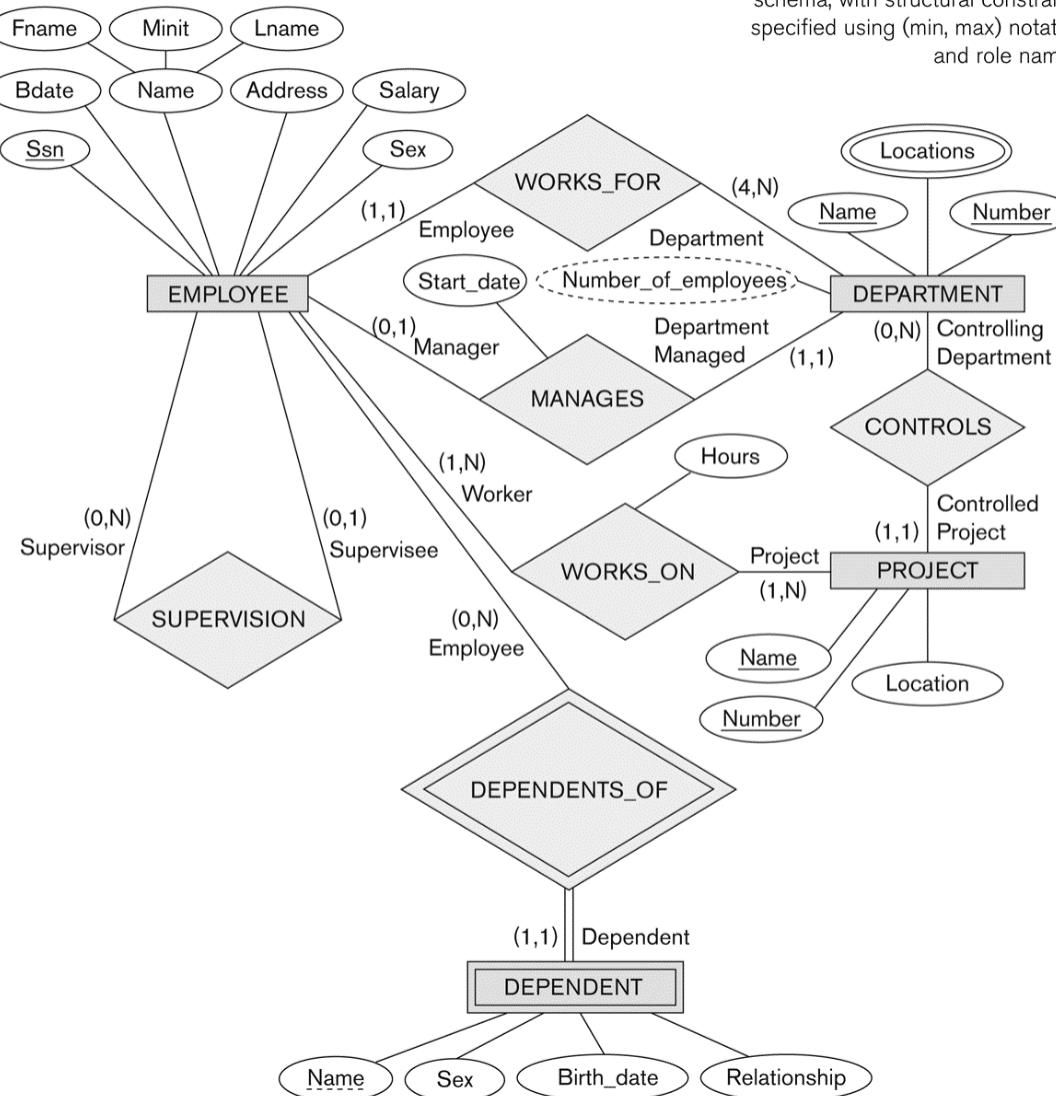


Read the min,max numbers next to the entity type and looking **away from** the entity type

# COMPANY ER Schema Diagram using (min, max) notation

**Figure 3.15**

ER diagrams for the company schema, with structural constraints specified using (min, max) notation and role names.



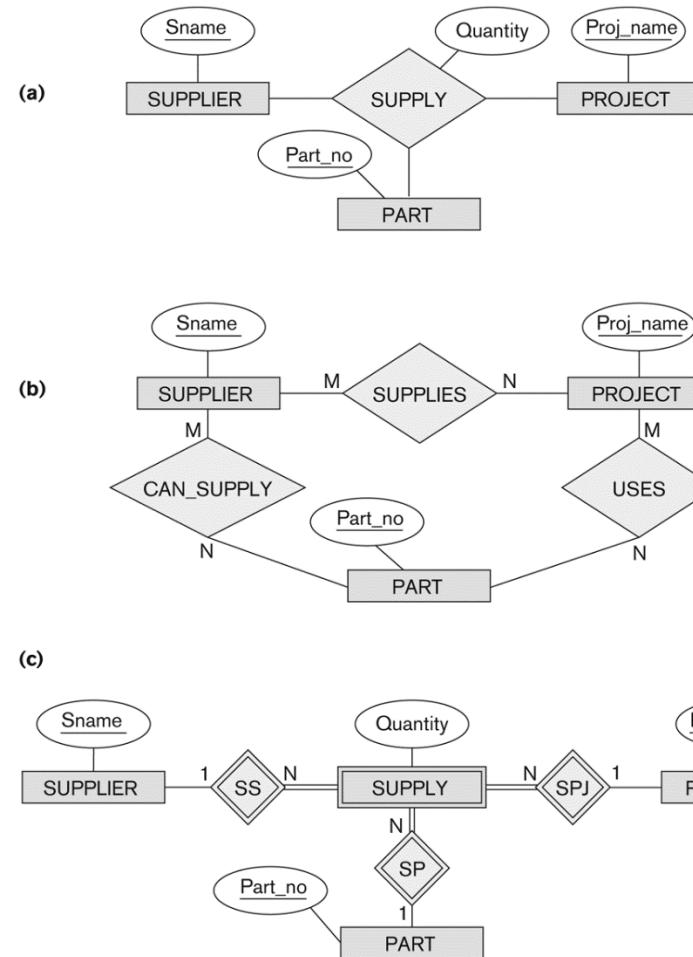
# Relationships of Higher Degree

- Relationship types of degree 2 are called binary
- Relationship types of degree 3 are called ternary and of degree n are called n-ary
- 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

# Discussion of n-ary relationships ( $n > 2$ )

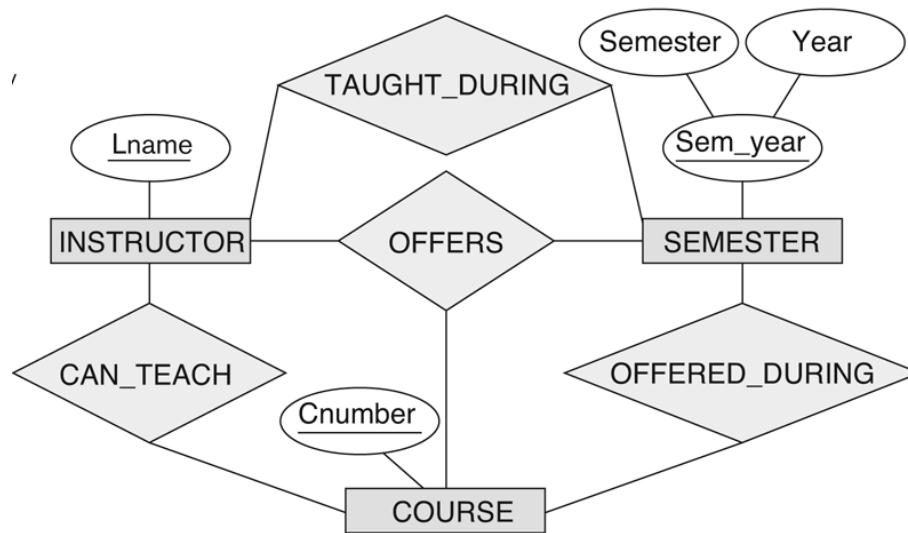
- In general, 3 binary relationships can represent different information than a single ternary relationship (see Figure 3.17a and b on next slide)
- If needed, the binary and n-ary relationships can all be included in the schema design (see Figure 3.17a and b, where all relationships convey different meanings)
- In some cases, a ternary relationship can be represented as a weak entity if the data model allows a weak entity type to have multiple identifying relationships (and hence multiple owner entity types) (see Figure 3.17c)

# Example of a ternary relationship



# Discussion of n-ary relationships ( $n > 2$ )

- If a particular binary relationship can be derived from a higher-degree relationship at all times, then it is redundant and therefore, can be left out.
- For example, the TAUGHT\_DURING and OFFERED\_DURING binary relationship in Figure 3.18 (see next slide) can be derived from the ternary relationship OFFERS (based on the meaning of the relationships), so they can be removed.



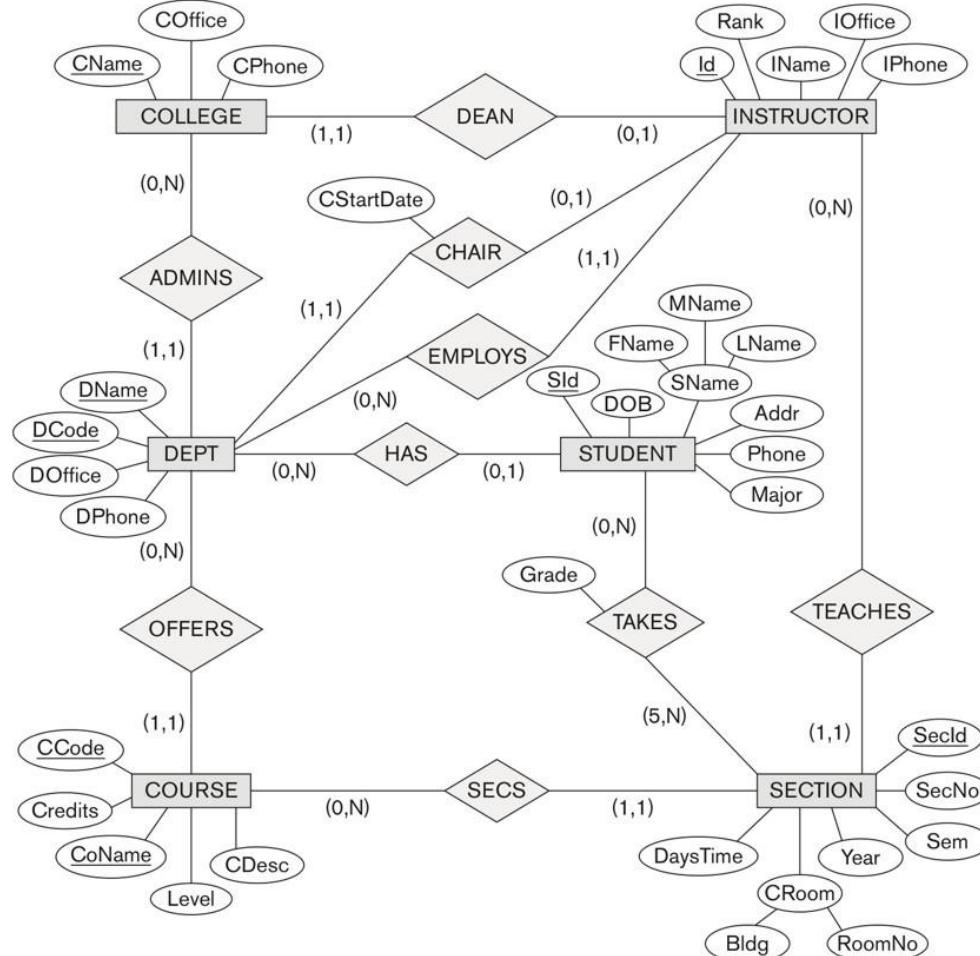
# Displaying constraints on higher-degree relationships

- The (min, max) constraints can be displayed on the edges – however, they do not fully describe the constraints
- Displaying a 1, M, or N indicates additional constraints
  - An M or N indicates no constraint
  - A 1 indicates that an entity can participate in at most one relationship instance *that has a particular combination of the other participating entities*
- In general, both (min, max) and 1, M, or N are needed to describe fully the constraints
- Overall, the constraint specification is difficult and possibly ambiguous when we consider relationships of a degree higher than two.

# Another Example: A UNIVERSITY Database

- To keep track of the enrollments in classes and student grades, another database is to be designed.
- It keeps track of the COLLEGES, DEPARTMENTS within each college, the COURSES offered by departments, and SECTIONS of courses, INSTRUCTORS who teach the sections etc.
- These entity types and the relationships among these entity types are shown on the next slide in Figure 3.20.

# UNIVERSITY database conceptual schema



# Extended Entity-Relationship (EER) Model

- The entity relationship model in its original form did not support the specialization and generalization abstractions
- Next, we will see how the ER model can be extended with
  - Type-subtype and set-subset relationships
  - Specialization/Generalization Hierarchies
  - Notation to display them in EER diagrams

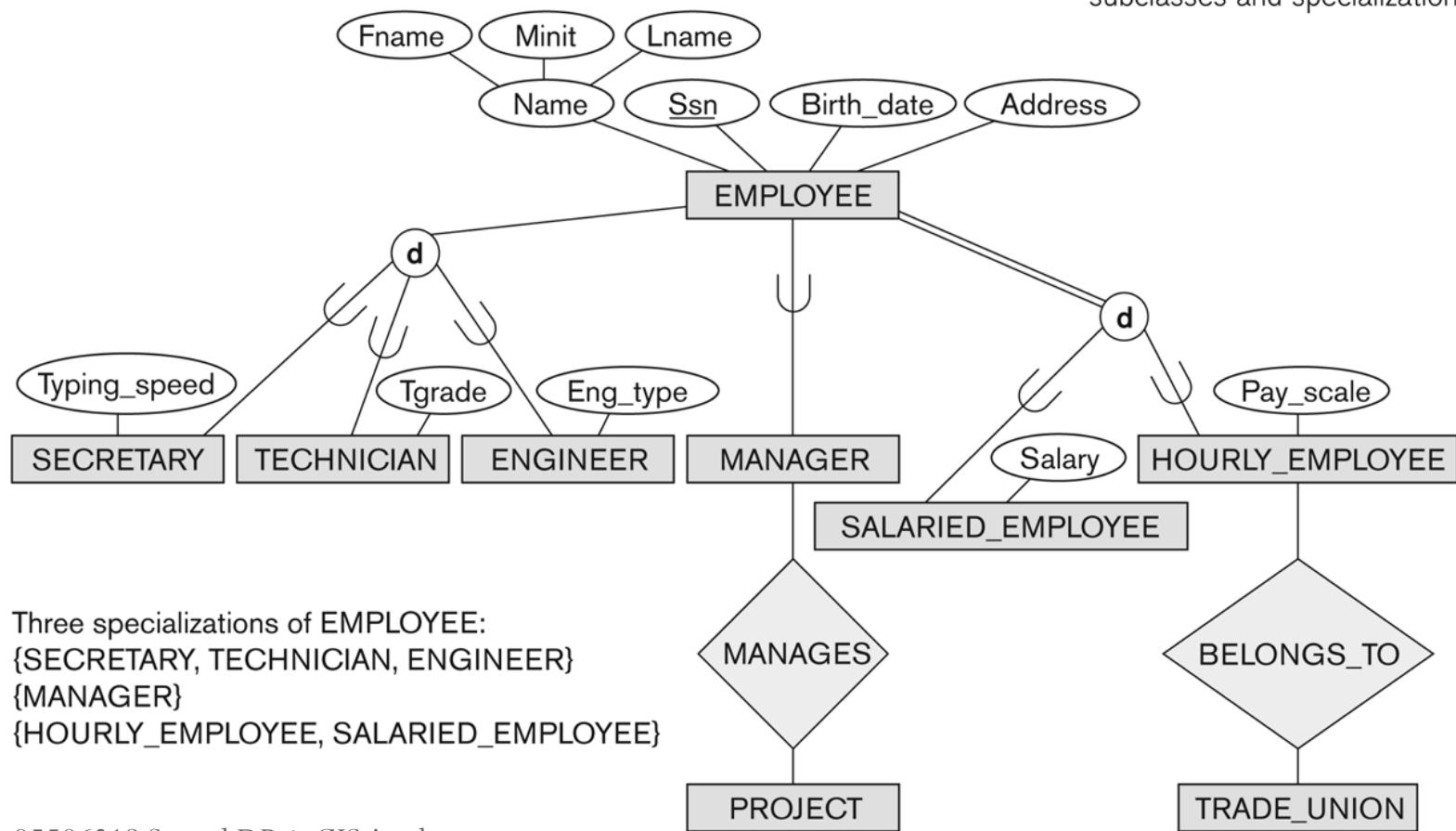
# Subclasses and Superclasses (1)

- An entity type may have additional meaningful subgroupings of its entities
  - Example: EMPLOYEE may be further grouped into:
    - SECRETARY, ENGINEER, TECHNICIAN, ...
      - Based on the EMPLOYEE's Job
    - MANAGER
      - EMPLOYEES who are managers (the role they play)
    - SALARIED\_EMPLOYEE, HOURLY\_EMPLOYEE
      - Based on the EMPLOYEE's method of pay
  - EER diagrams extend ER diagrams to represent these additional subgroupings, called *subclasses* or *subtypes*

# Subclasses and Superclasses

**Figure 4.1**

EER diagram notation to represent subclasses and specialization.



# Subclasses and Superclasses (2)

- Each of these subgroupings is a subset of EMPLOYEE entities
- Each is called a subclass of EMPLOYEE
- EMPLOYEE is the superclass for each of these subclasses
- These are called superclass/subclass relationships:
  - EMPLOYEE/SECRETARY
  - EMPLOYEE/TECHNICIAN
  - EMPLOYEE/MANAGER
  - ...

# Subclasses and Superclasses (3)

- These are also called IS-A relationships
  - SECRETARY IS-A EMPLOYEE, TECHNICIAN IS-A EMPLOYEE,
    - ....
- Note: An entity that is member of a subclass represents the same real-world entity as some member of the superclass:
  - The subclass member is the same entity in a *distinct specific role*
  - An entity cannot exist in the database merely by being a member of a subclass; it must also be a member of the superclass
  - A member of the superclass can be optionally included as a member of any number of its subclasses

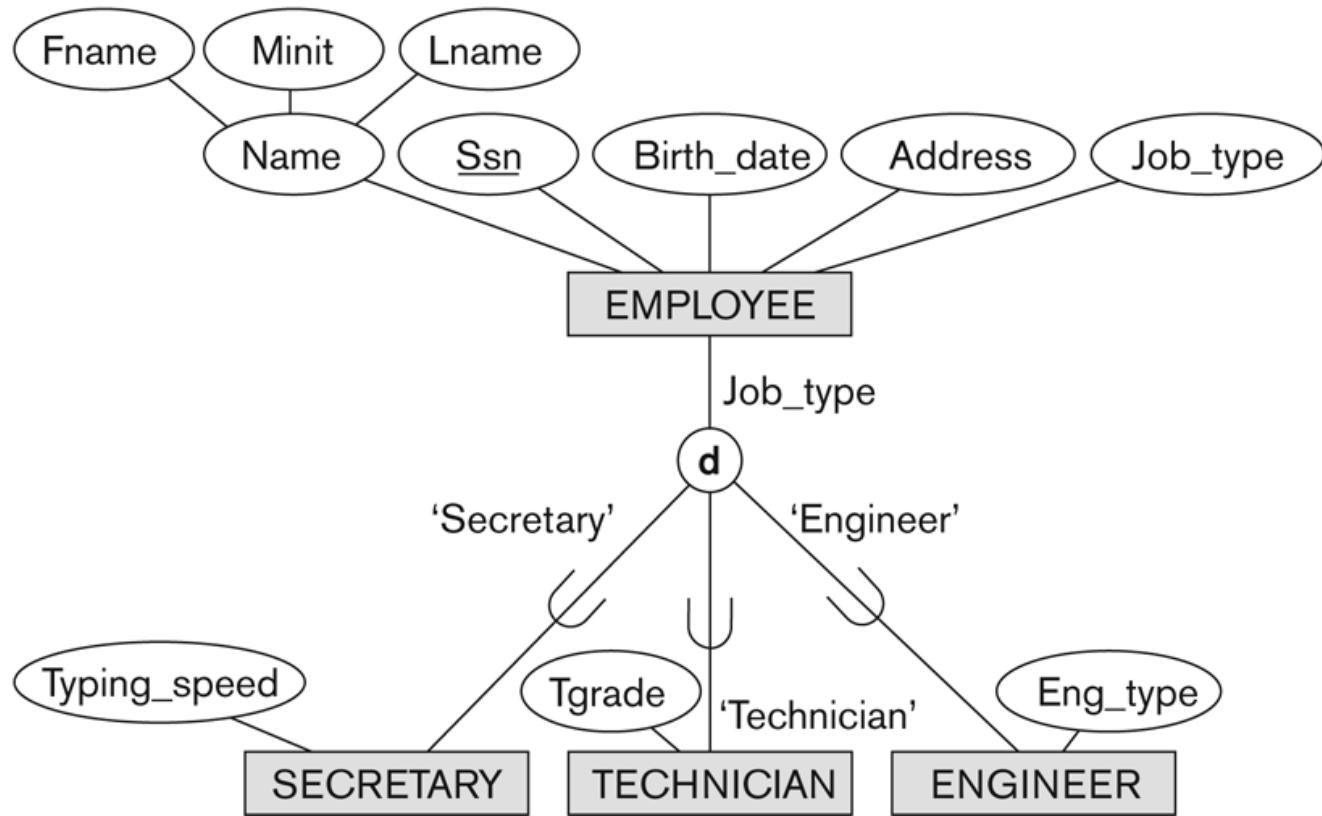
# Subclasses and Superclasses (4)

- Examples:
  - A salaried employee who is also an engineer belongs to the two subclasses:
    - ENGINEER, and
    - SALARIED\_EMPLOYEE
  - A salaried employee who is also an engineering manager belongs to the three subclasses:
    - MANAGER,
    - ENGINEER, and
    - SALARIED\_EMPLOYEE
  - It is not necessary that every entity in a superclass be a member of some subclass

# Representing Specialization in EER Diagrams

**Figure 4.4**

EER diagram notation for an attribute-defined specialization on Job\_type.



# Attribute Inheritance in Superclass / Subclass Relationships

- An entity that is member of a subclass *inherits*
  - All attributes of the entity as a member of the superclass
  - All relationships of the entity as a member of the superclass
- Example:
  - In the previous slide, SECRETARY (as well as TECHNICIAN and ENGINEER) inherit the attributes Name, SSN, ..., from EMPLOYEE
  - Every SECRETARY entity will have values for the inherited attributes

# Specialization (1)

- Specialization is the process of defining a set of subclasses of a superclass
- The set of subclasses is based upon some distinguishing characteristics of the entities in the superclass
  - Example: {SECRETARY, ENGINEER, TECHNICIAN} is a specialization of EMPLOYEE based upon *job type*.
  - Example: MANAGER is a specialization of EMPLOYEE based on the role the employee plays
    - May have several specializations of the same superclass

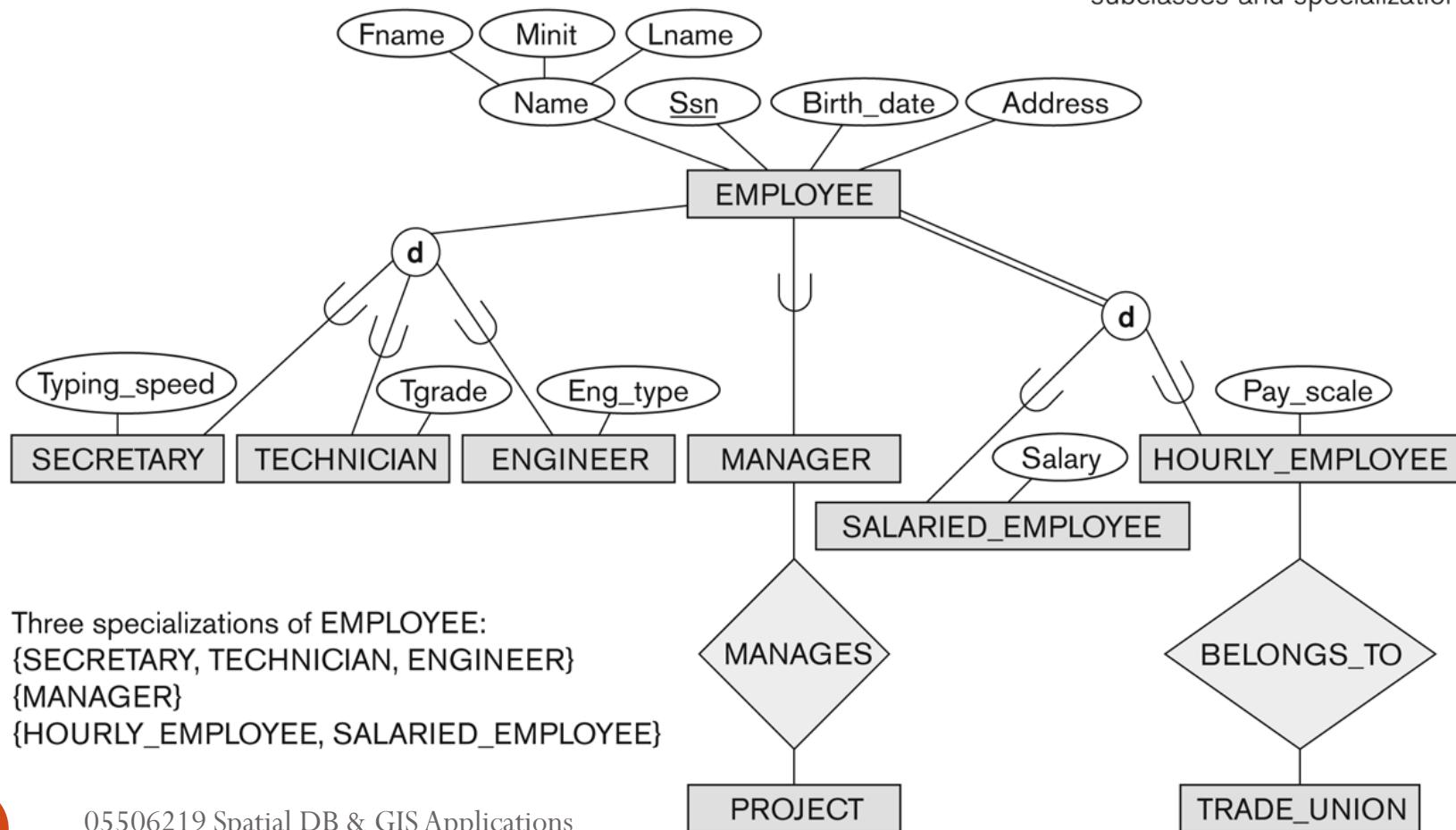
## Specialization (2)

- Example: Another specialization of EMPLOYEE based on *method of pay* is {SALARIED\_EMPLOYEE, HOURLY\_EMPLOYEE}.
  - Superclass/subclass relationships and specialization can be diagrammatically represented in EER diagrams
  - Attributes of a subclass are called *specific* or *local* attributes.
    - For example, the attribute TypingSpeed of SECRETARY
  - The subclass can also participate in specific relationship types.
    - For example, a relationship BELONGS\_TO of HOURLY\_EMPLOYEE

# Specialization (3)

**Figure 4.1**

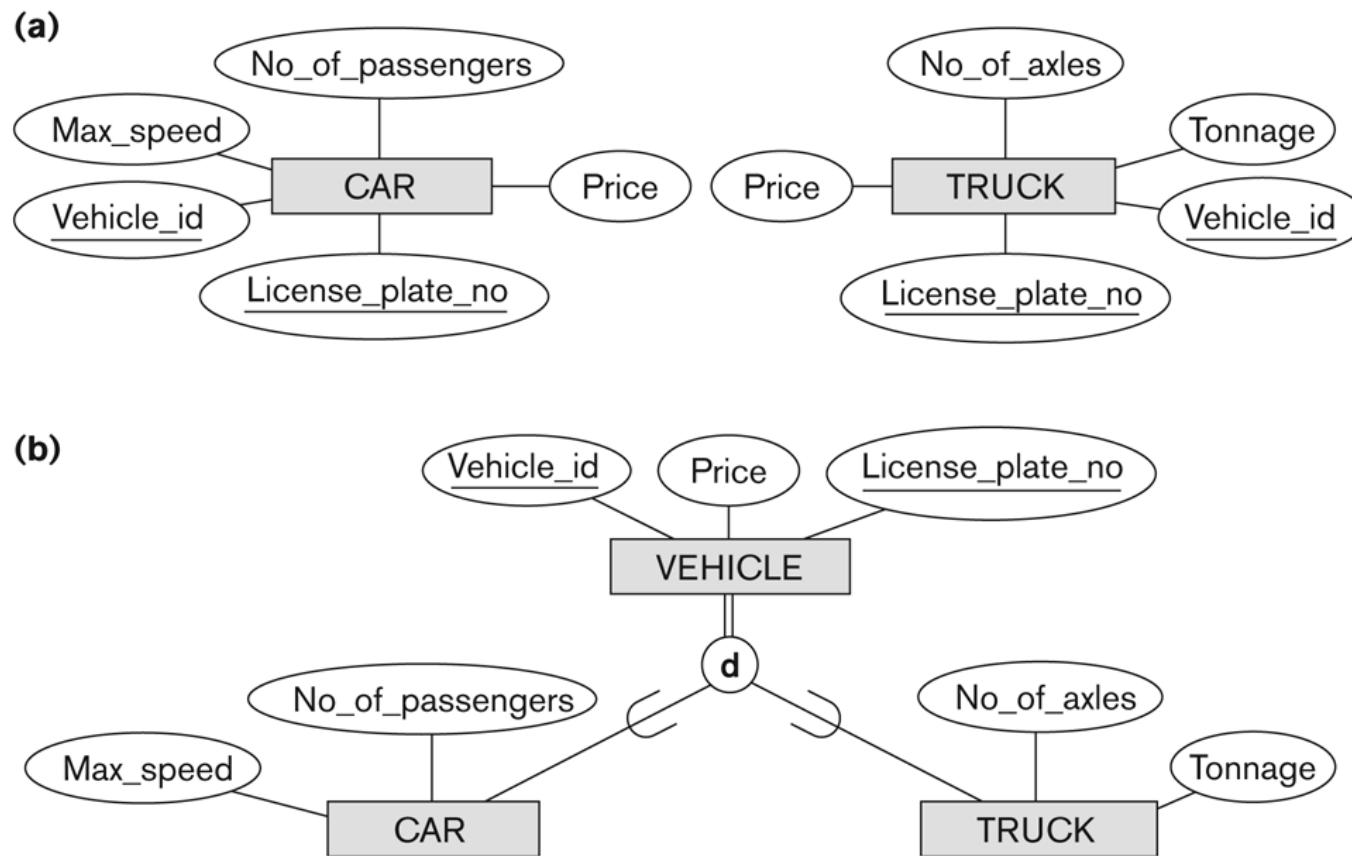
EER diagram notation to represent subclasses and specialization.



# Generalization

- Generalization is the reverse of the specialization process
- Several classes with common features are generalized into a superclass;
  - original classes become its subclasses
- Example: CAR, TRUCK generalized into VEHICLE;
  - both CAR, TRUCK become subclasses of the superclass VEHICLE.
  - We can view {CAR, TRUCK} as a specialization of VEHICLE
  - Alternatively, we can view VEHICLE as a generalization of CAR and TRUCK

# Generalization (2)



**Figure 4.3**  
Generalization. (a) Two entity types, CAR and TRUCK.  
(b) Generalizing CAR and TRUCK into the superclass VEHICLE.

# Generalization and Specialization (1)

- Diagrammatic notations are sometimes used to distinguish between generalization and specialization
  - Arrow pointing to the generalized superclass represents a generalization
  - Arrows pointing to the specialized subclasses represent a specialization
  - We *do not use* this notation because it is often subjective as to which process is more appropriate for a particular situation
  - We advocate not drawing any arrows

# Generalization and Specialization (2)

- Data Modeling with Specialization and Generalization
  - A superclass or subclass represents a collection (or set or grouping) of entities
  - It also represents a particular *type of entity*
  - Shown in rectangles in EER diagrams (as are entity types)
  - We can call all entity types (and their corresponding collections) *classes*, whether they are entity types, superclasses, or subclasses

# Types of Specialization

- Predicate-defined ( or condition-defined) : based on some predicate. E.g., based on value of an attribute, say, Job-type, or Age.
- Attribute-defined: shows the name of the attribute next to the line drawn from the superclass toward the subclasses (see Fig. 4.1)
- User-defined: membership is defined by the user on an entity by entity basis

# Constraints on Specialization and Generalization (1)

- If we can determine exactly those entities that will become members of each subclass by a condition, the subclasses are called predicate-defined (or condition-defined) subclasses
  - Condition is a constraint that determines subclass members
  - Display a predicate-defined subclass by writing the predicate condition next to the line attaching the subclass to its superclass

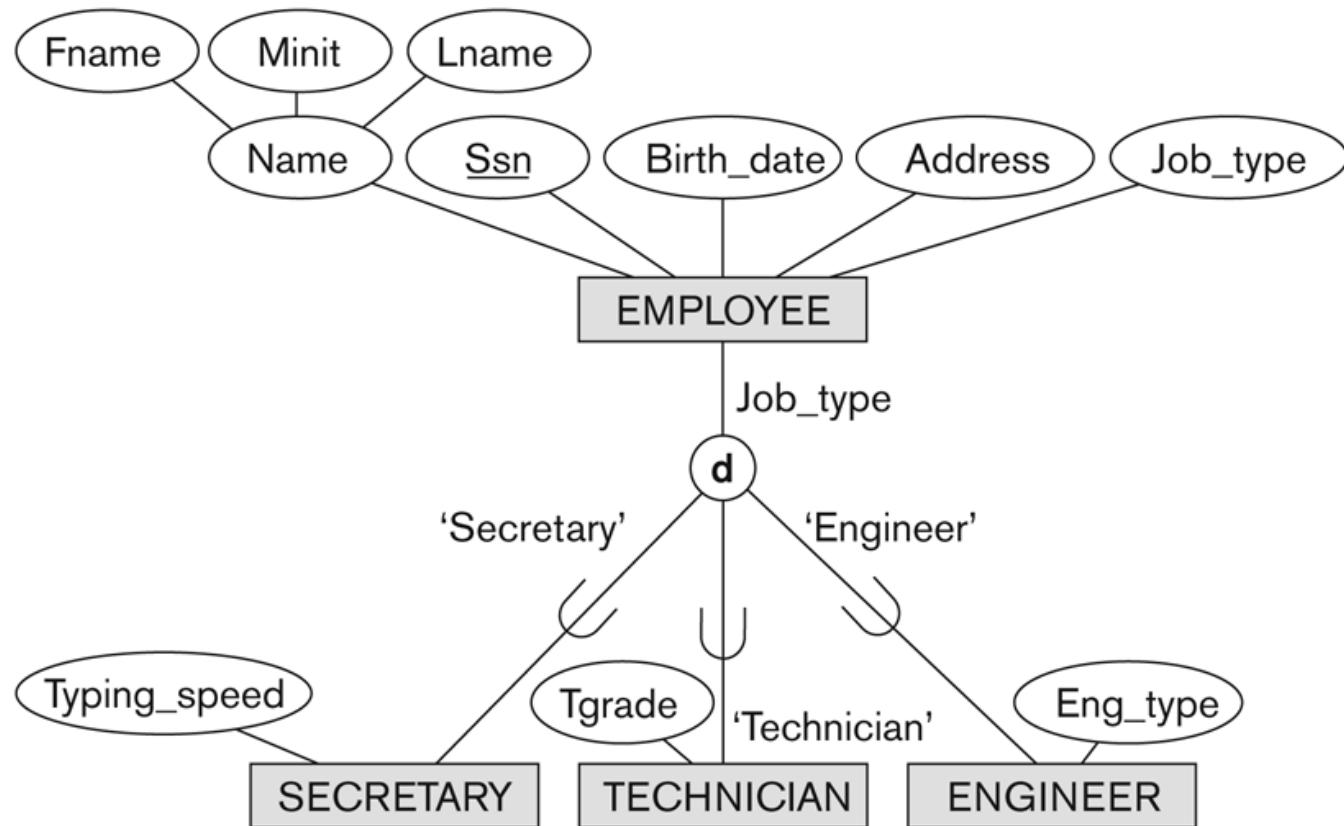
# Constraints on Specialization and Generalization (2)

- If all subclasses in a specialization have membership condition on same attribute of the superclass, specialization is called an attribute-defined specialization
  - Attribute is called the defining attribute of the specialization
  - Example: JobType is the defining attribute of the specialization {SECRETARY, TECHNICIAN, ENGINEER} of EMPLOYEE
- If no condition determines membership, the subclass is called user-defined
  - Membership in a subclass is determined by the database users by applying an operation to add an entity to the subclass
  - Membership in the subclass is specified individually for each entity in the superclass by the user

# Displaying an attribute-defined specialization in EER diagrams

**Figure 4.4**

EER diagram notation for an attribute-defined specialization on Job\_type.



# Constraints on Specialization and Generalization (3)

- Two basic constraints can apply to a specialization/generalization:
  - Disjointness Constraint:
  - Completeness Constraint:

# Constraints on Specialization and Generalization (4)

- Disjointness Constraint:
  - Specifies that the subclasses of the specialization must be *disjoint*:
    - an entity can be a member of at most one of the subclasses of the specialization
  - Specified by d in EER diagram
  - If not disjoint, specialization is *overlapping*:
    - that is the same entity may be a member of more than one subclass of the specialization
  - Specified by o in EER diagram

# Constraints on Specialization and Generalization (5)

- Completeness (Exhaustiveness) Constraint:
  - *Total* specifies that every entity in the superclass must be a member of some subclass in the specialization/generalization
  - Shown in EER diagrams by a **double line**
  - *Partial* allows an entity not to belong to any of the subclasses
  - Shown in EER diagrams by a single line

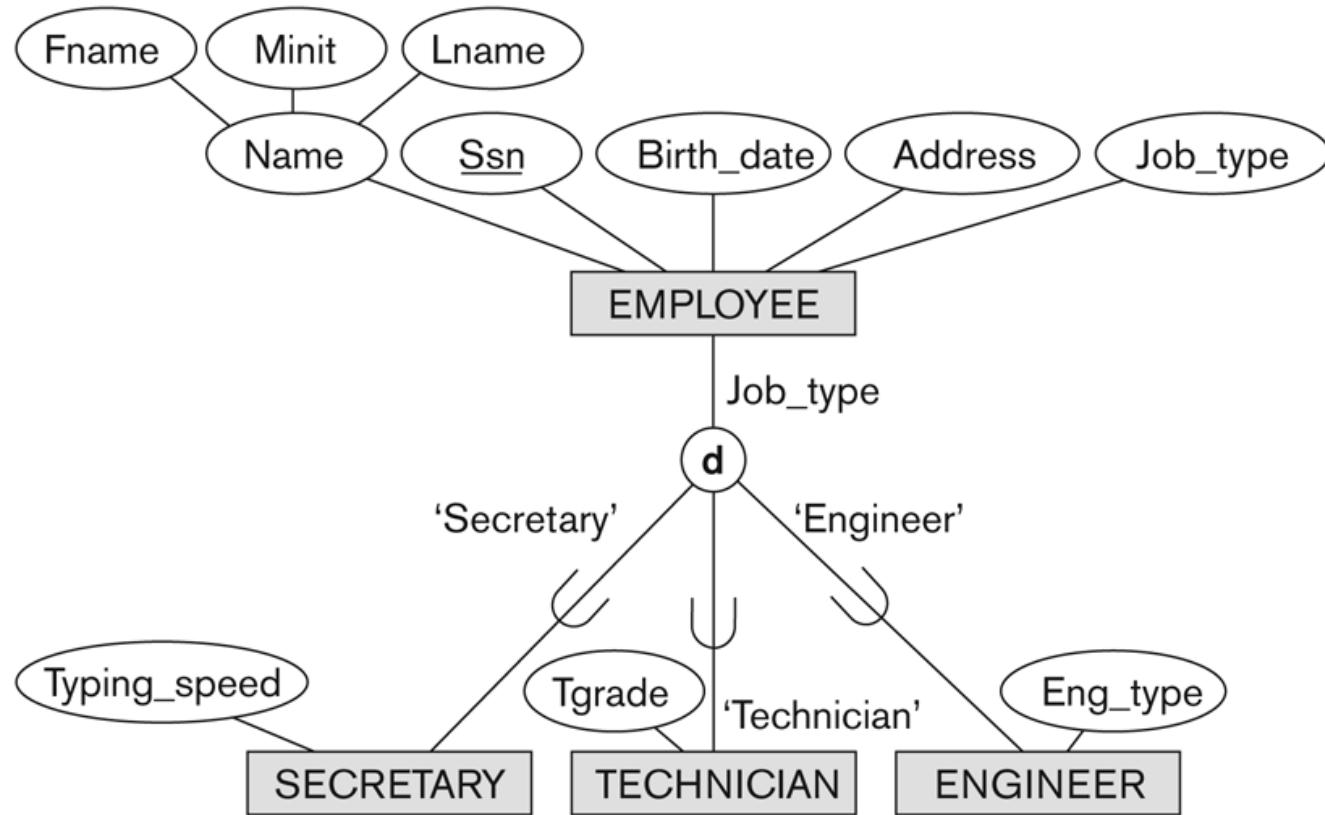
# Constraints on Specialization and Generalization (6)

- Hence, we have four types of specialization/generalization:
  - Disjoint, total
  - Disjoint, partial
  - Overlapping, total
  - Overlapping, partial
- Note: Generalization usually is total because the superclass is derived from the subclasses.

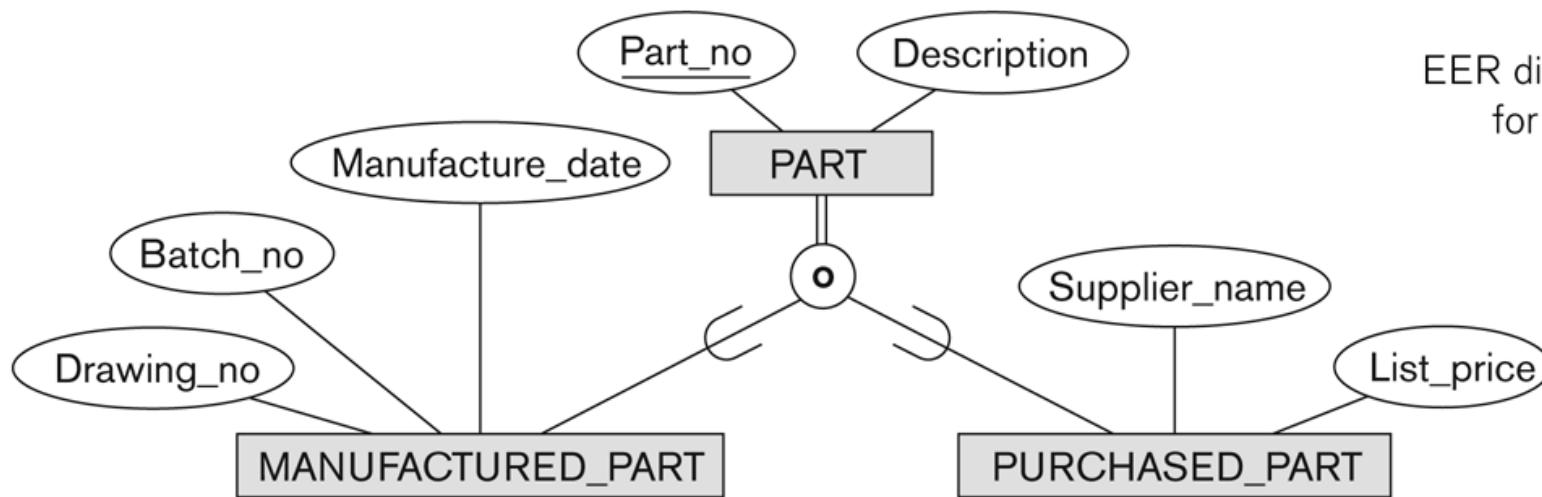
# Example of disjoint partial Specialization

**Figure 4.4**

EER diagram notation for an attribute-defined specialization on Job\_type.



# Example of overlapping total Specialization



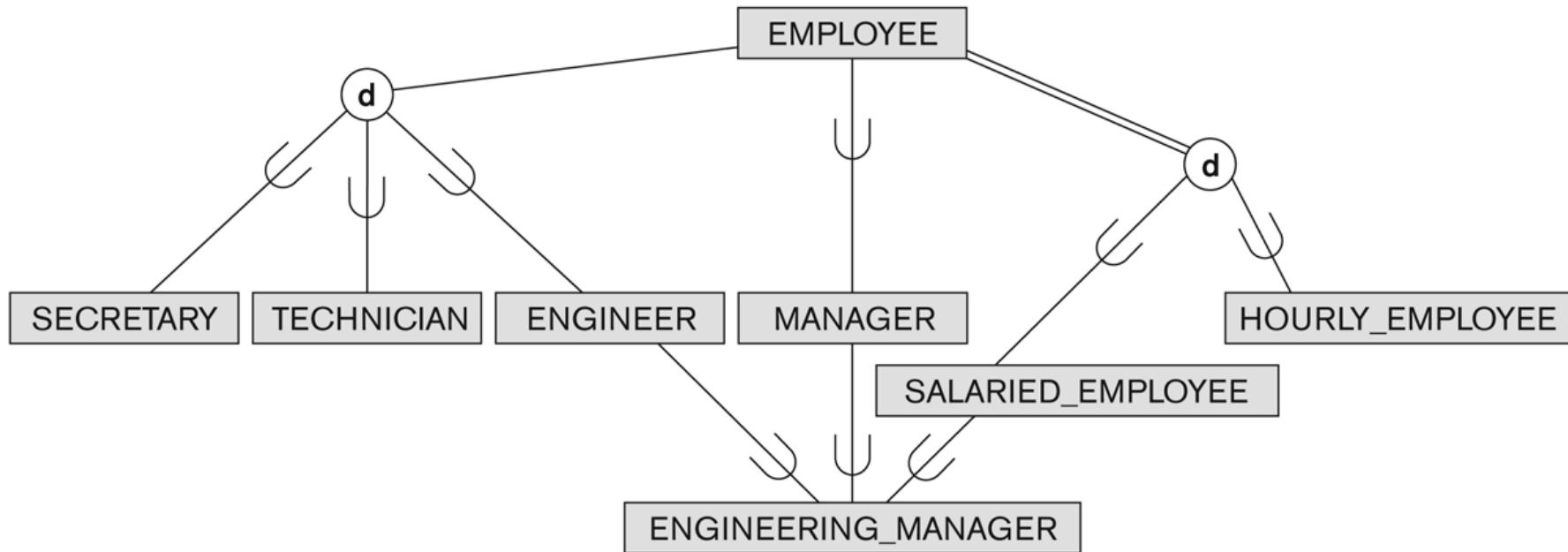
**Figure 4.5**

EER diagram notation  
for an overlapping  
(nondisjoint)  
specialization.

# Specialization/Generalization Hierarchies, Lattices & Shared Subclasses (1)

- A subclass may itself have further subclasses specified on it
  - forms a hierarchy or a lattice
- **Hierarchy** has a constraint that every subclass has only one superclass (called *single inheritance*); this is basically a **tree structure**
- In a **lattice**, a subclass can be subclass of more than one superclass (called **multiple inheritance**)

## Shared Subclass “Engineering\_Manager”



**Figure 4.6**  
A specialization lattice with shared subclass **ENGINEERING\_MANAGER**.

# Specialization/Generalization Hierarchies, Lattices & Shared Subclasses (2)

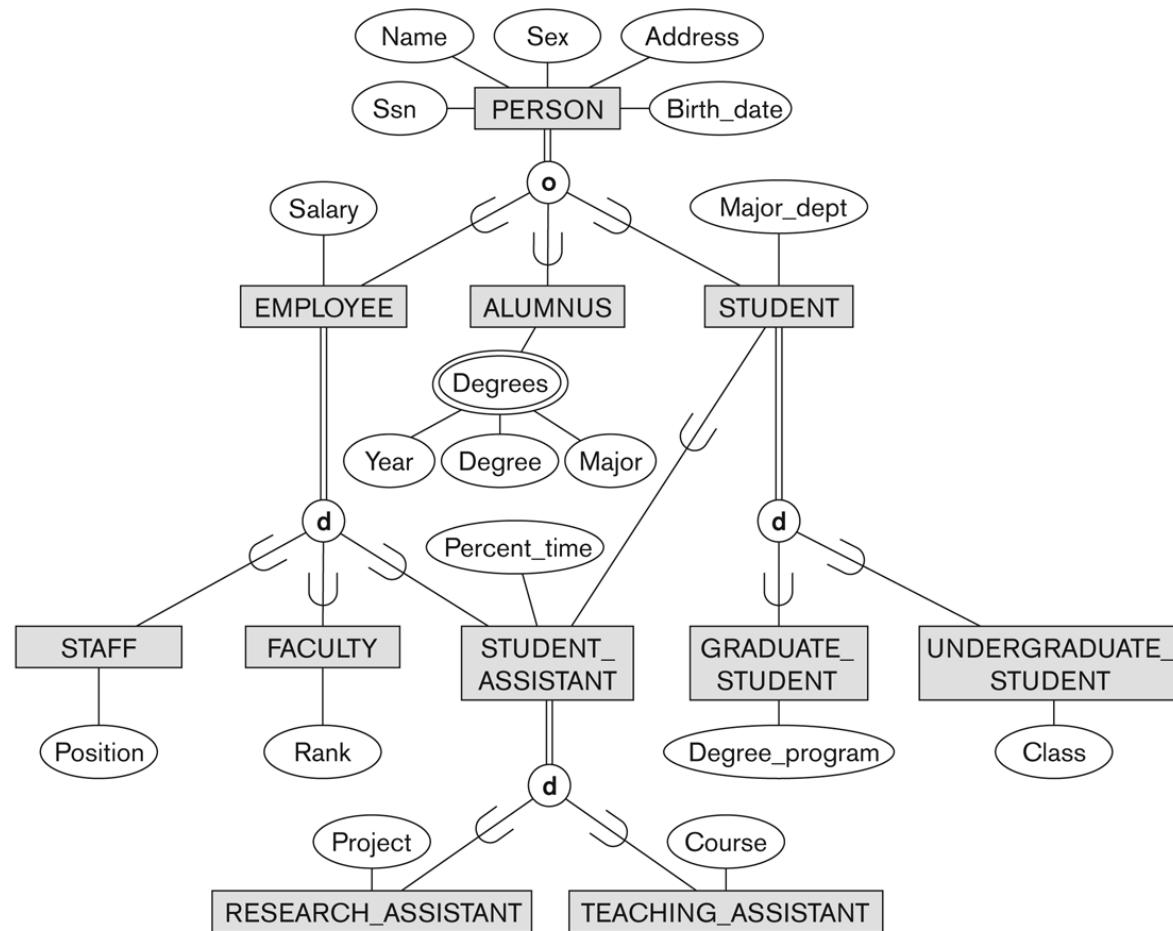
- In a lattice or hierarchy, a subclass inherits attributes not only of its direct superclass, but also of all its predecessor superclasses
- A subclass with more than one superclass is called a shared subclass (multiple inheritance)
- Can have:
  - *specialization* hierarchies or lattices, or
  - *generalization* hierarchies or lattices,
  - depending on how they were *derived*
- We just use *specialization* (to stand for the end result of either specialization or generalization)

# Specialization/Generalization Hierarchies, Lattices & Shared Subclasses (3)

- In *specialization*, start with an entity type and then define subclasses of the entity type by successive specialization
  - called a *top down* conceptual refinement process
- In *generalization*, start with many entity types and generalize those that have common properties
  - Called a *bottom up* conceptual synthesis process
- In practice, a *combination of both processes* is usually employed

# Specialization / Generalization Lattice

## Example (UNIVERSITY)



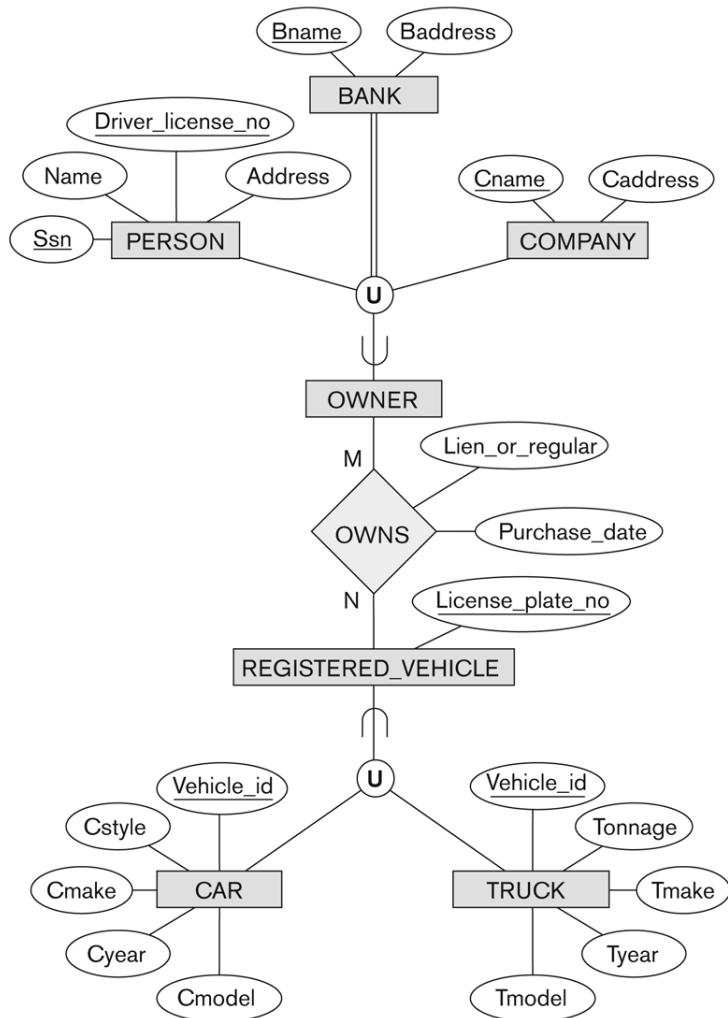
# Categories (UNION TYPES) (1)

- All of the *superclass/subclass relationships* we have seen thus far have a single superclass
- A shared subclass is a subclass in:
  - *more than one* distinct superclass/subclass relationships
  - each relationships has a *single* superclass
  - shared subclass leads to multiple inheritance
- In some cases, we need to model a *single superclass/subclass relationship* with more than one superclass
- Superclasses can represent different entity types
- Such a subclass is called a category or UNION TYPE

# Categories (UNION TYPES) (2)

- Example: In a database for vehicle registration, a vehicle owner can be a PERSON, a BANK (holding a lien on a vehicle) or a COMPANY.
  - A *category* (UNION type) called OWNER is created to represent a subset of the *union* of the three superclasses COMPANY, BANK, and PERSON
  - A category member must exist in *at least one (typically just one)* of its superclasses
- Difference from *shared subclass*, which is a:
  - subset of the *intersection* of its superclasses
  - shared subclass member must exist in *all* of its superclasses

# Two categories (UNION types): OWNER, REGISTERED\_VEHICLE



**Figure 4.8**

Two categories (union types): OWNER and REGISTERED\_VEHICLE.

# Chapter 5: Logical Database Design (ER- and EER-to-Relational Mapping)



คณะวิทยาศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง  
Faculty of Science, King Mongkut's Institute of Technology Ladkrabang

# Outline

- **ER-to-Relational Mapping Algorithm**
  - Step 1: Mapping of Regular Entity Types
  - Step 2: Mapping of Weak Entity Types
  - Step 3: Mapping of Binary 1:1 Relation Types
  - Step 4: Mapping of Binary 1:N Relationship Types.
  - Step 5: Mapping of Binary M:N Relationship Types.
  - Step 6: Mapping of Multivalued attributes.
  - Step 7: Mapping of N-ary Relationship Types.
- **Mapping EER Model Constructs to Relations**
  - Step 8: Options for Mapping Specialization or Generalization.
  - Step 9: Mapping of Union Types (Categories).

# GOALS during Mapping

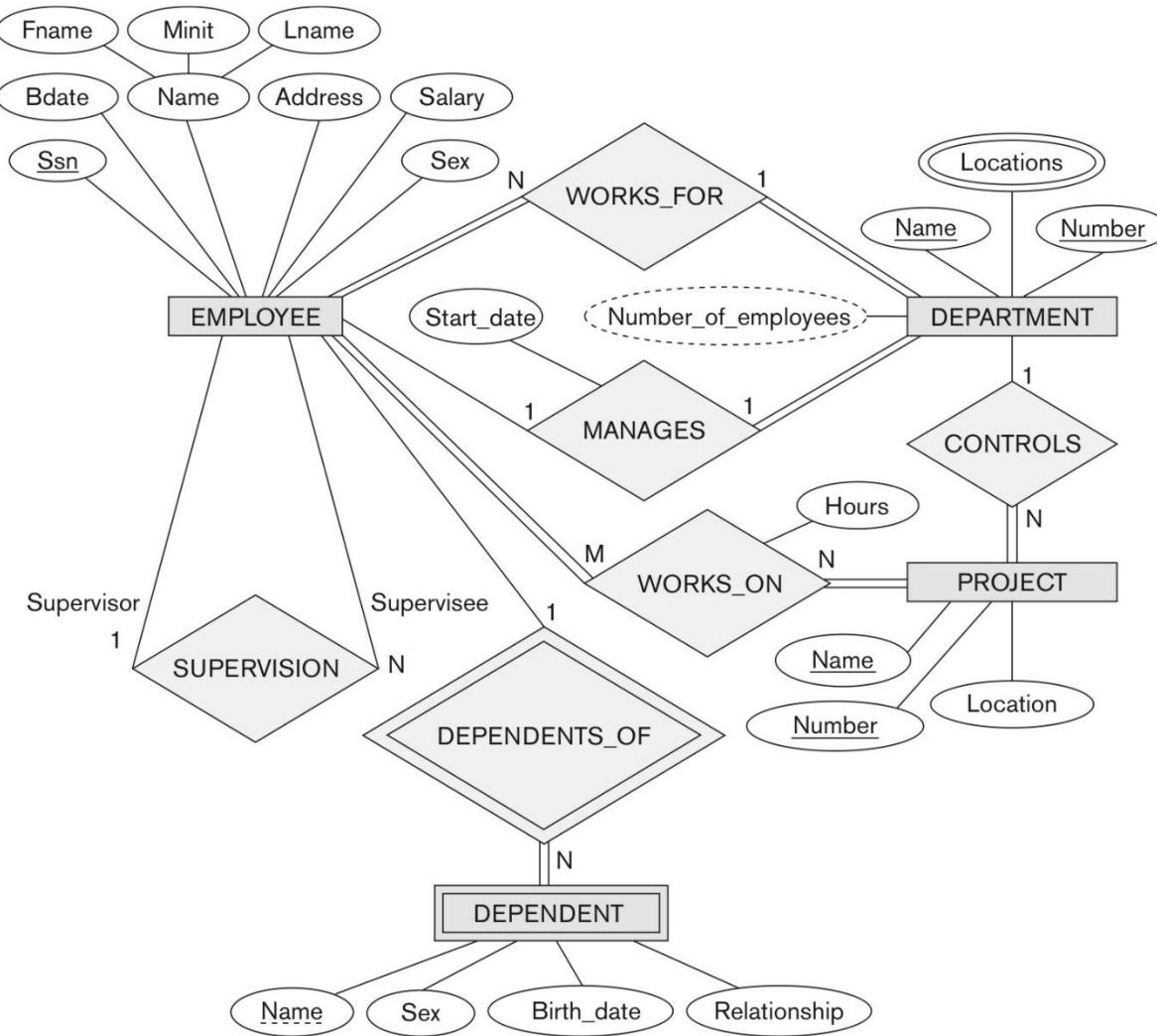
- Preserve all information (that includes all attributes)
- Maintain the constraints to the extent possible (Relational Model cannot preserve all constraints- e.g., max cardinality ratio such as 1:10 in ER; exhaustive classification into subtypes, e.g., STUDENTS are specialized into Domestic and Foreign)
- Minimize null values

*The mapping procedure described has been implemented in many commercial tools.*

# ER-to-Relational Mapping Algorithm

- Step 1: Mapping of Regular Entity Types.
  - For each regular (strong) entity type E in the ER schema, create a relation R that includes all the simple attributes of E.
  - Choose one of the key attributes of E as the primary key for R.
  - If the chosen key of E is composite, the set of simple attributes that form it will together form the primary key of R.
- Example: We create the relations EMPLOYEE, DEPARTMENT, and PROJECT in the relational schema corresponding to the regular entities in the ER diagram.
  - SSN, DNUMBER, and PNUMBER are the primary keys for the relations EMPLOYEE, DEPARTMENT, and PROJECT as shown.

# Figure 9.1 The ER conceptual schema diagram for the COMPANY database.



## ER-to-Relational Mapping Algorithm (contd.)

- **Step 2: Mapping of Weak Entity Types**
  - For each weak entity type W in the ER schema with owner entity type E, create a relation R & include all simple attributes (or simple components of composite attributes) of W as attributes of R.
  - Also, include as foreign key attributes of R the primary key attribute(s) of the relation(s) that correspond to the owner entity type(s).
  - The primary key of R is the *combination* of the primary key(s) of the owner(s) and the partial key of the weak entity type W, if any.
- **Example:** Create the relation DEPENDENT in this step to correspond to the weak entity type DEPENDENT.
  - Include the primary key SSN of the EMPLOYEE relation as a foreign key attribute of DEPENDENT (renamed to ESSN).
  - The primary key of the DEPENDENT relation is the combination {ESSN, DEPENDENT\_NAME} because DEPENDENT\_NAME is the partial key of DEPENDENT.

# ER-to-Relational Mapping Algorithm (contd.)

- **Step 3: Mapping of Binary 1:1 Relation Types**
  - For each binary 1:1 relationship type R in the ER schema, identify the relations S and T that correspond to the entity types participating in R.
- There are three possible approaches:
  1. **Foreign Key ( 2 relations) approach:** Choose one of the relations-say S-and include a foreign key in S the primary key of T. It is better to choose an entity type with total participation in R in the role of S.
    - Example: 1:1 relation MANAGES is mapped by choosing the participating entity type DEPARTMENT to serve in the role of S, because its participation in the MANAGES relationship type is total.
  2. **Merged relation (1 relation) option:** An alternate mapping of a 1:1 relationship type is possible by merging the two entity types and the relationship into a single relation. This may be appropriate when both participations are total.
  3. **Cross-reference or relationship relation ( 3 relations) option:** The third alternative is to 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.

## ER-to-Relational Mapping Algorithm (contd.)

- Step 4: Mapping of Binary 1:N Relationship Types.
  - For each regular binary 1:N relationship type R, identify the relation S that represent the participating entity type at the N-side of the relationship type.
  - Include as foreign key in S the primary key of the relation T that represents the other entity type participating in R.
  - Include any simple attributes of the 1:N relation type as attributes of S.
- Example: 1:N relationship types WORKS\_FOR, CONTROLS, and SUPERVISION in the figure.
  - For WORKS\_FOR we include the primary key DNUMBER of the DEPARTMENT relation as foreign key in the EMPLOYEE relation and call it DNO.
  - An alternative approach is to use a Relationship relation (cross referencing relation) – this is rarely done.

## ER-to-Relational Mapping Algorithm (contd.)

- **Step 5: Mapping of Binary M:N Relationship Types.**
  - For each regular binary M:N relationship type R, *create a new relation S to represent R. This is a relationship relation.*
  - Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types; *their combination will form the primary key of S.*
  - Also include any simple attributes of the M:N relationship type (or simple components of composite attributes) as attributes of S.
- Example: The M:N relationship type WORKS\_ON from the ER diagram is mapped by creating a relation WORKS\_ON in the relational database schema.
  - The primary keys of the PROJECT and EMPLOYEE relations are included as foreign keys in WORKS\_ON and renamed PNO and ESSN, respectively.
  - Attribute HOURS in WORKS\_ON represents the HOURS attribute of the relation type. The primary key of the WORKS\_ON relation is the combination of the foreign key attributes {ESSN, PNO}.

## ER-to-Relational Mapping Algorithm (contd.)

- **Step 6: Mapping of Multivalued attributes.**
  - For each multivalued attribute A, create a new relation R.
  - This relation R will include an attribute corresponding to A, plus the primary key attribute K-as a foreign key in R-of the relation that represents the entity type of relationship type that has A as an attribute.
  - The primary key of R is the combination of A and K. If the multivalued attribute is composite, we include its simple components.
- **Example:** The relation DEPT\_LOCATIONS is created.
  - The attribute DLOCATION represents the multivalued attribute LOCATIONS of DEPARTMENT, while DNUMBER-as foreign key-represents the primary key of the DEPARTMENT relation.
  - The primary key of R is the combination of {DNUMBER, DLOCATION}.

## ER-to-Relational Mapping Algorithm (contd.)

- **Step 7: Mapping of N-ary Relationship Types.**
  - For each n-ary relationship type R, where  $n > 2$ , create a new relationship S to represent R.
  - Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types.
  - Also include any simple attributes of the n-ary relationship type (or simple components of composite attributes) as attributes of S.
- **Example:** The relationship type SUPPY in the ER on the next slide.
  - This can be mapped to the relation SUPPLY shown in the relational schema, whose primary key is the combination of the three foreign keys {SNAME, PARTNO, PROJNAME}

# Figure 9.2 Result of mapping the COMPANY ER schema into a relational database schema.

## 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
-------	----------------	------------------	------

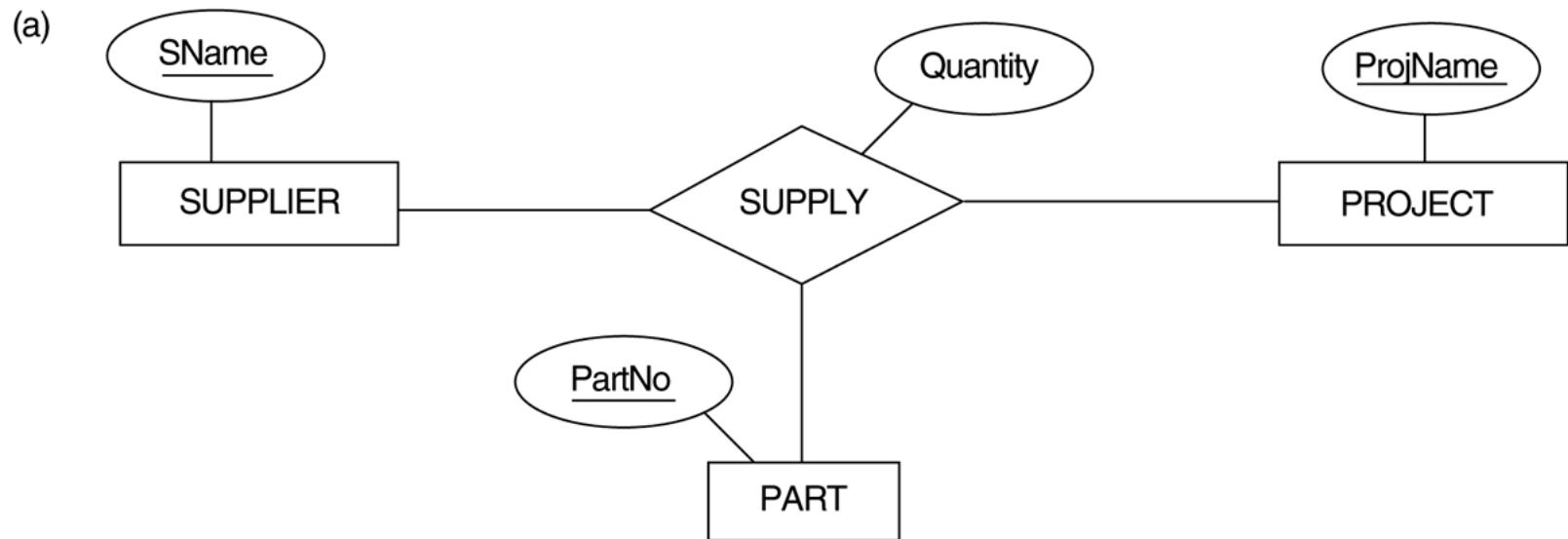
## WORKS\_ON

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

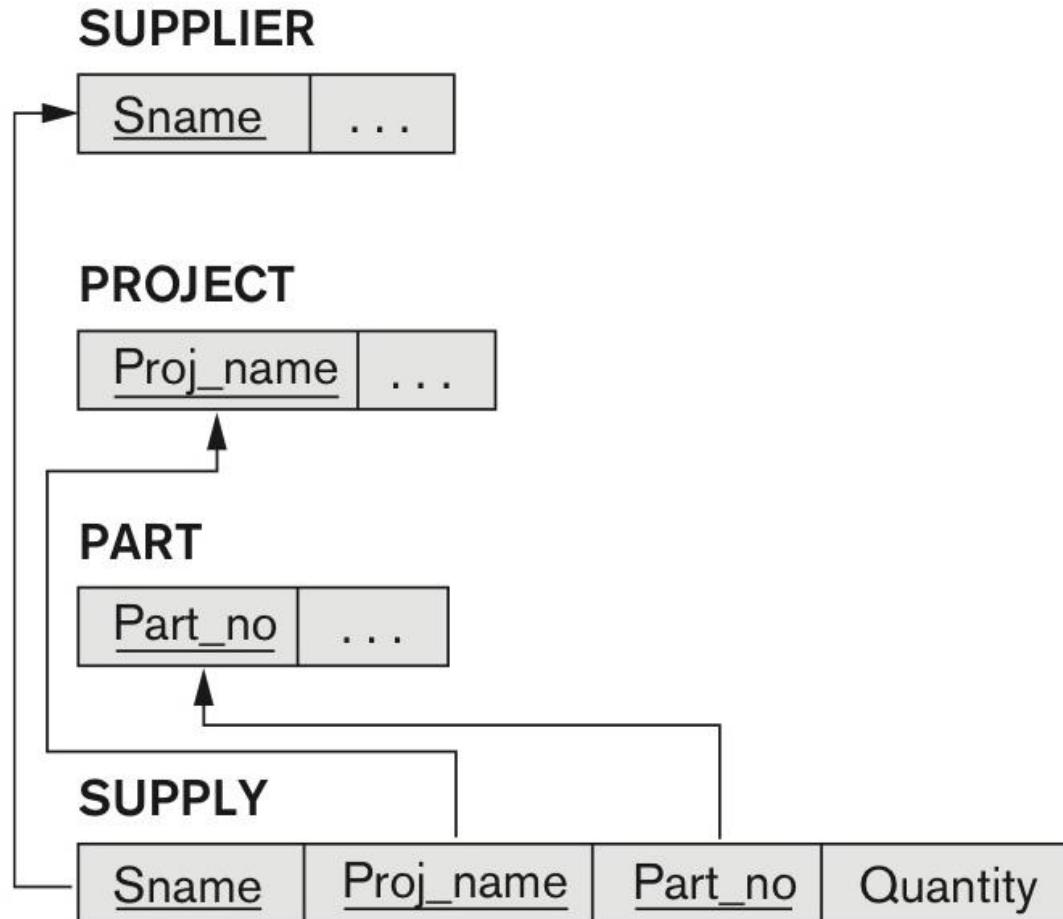
## DEPENDENT

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

## FIGURE 3.17 TERNARY RELATIONSHIP: SUPPLY



# Mapping the *n*-ary relationship type SUPPLY



# Summary of Mapping constructs and constraints

**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

# Mapping EER Model Constructs to Relations

- **Step8: Options for Mapping Specialization or Generalization.**
  - Convert each specialization with m subclasses {S1, S2, ..., Sm} and generalized superclass C, where the attributes of C are {k, a1, ..., an} and k is the (primary) key, into relational schemas using one of the four following options:
    - Option 8A: Multiple relations-Superclass and subclasses
    - Option 8B: Multiple relations-Subclass relations only
    - Option 8C: Single relation with one type attribute
    - Option 8D: Single relation with multiple type attributes

# Mapping EER Model Constructs to Relations

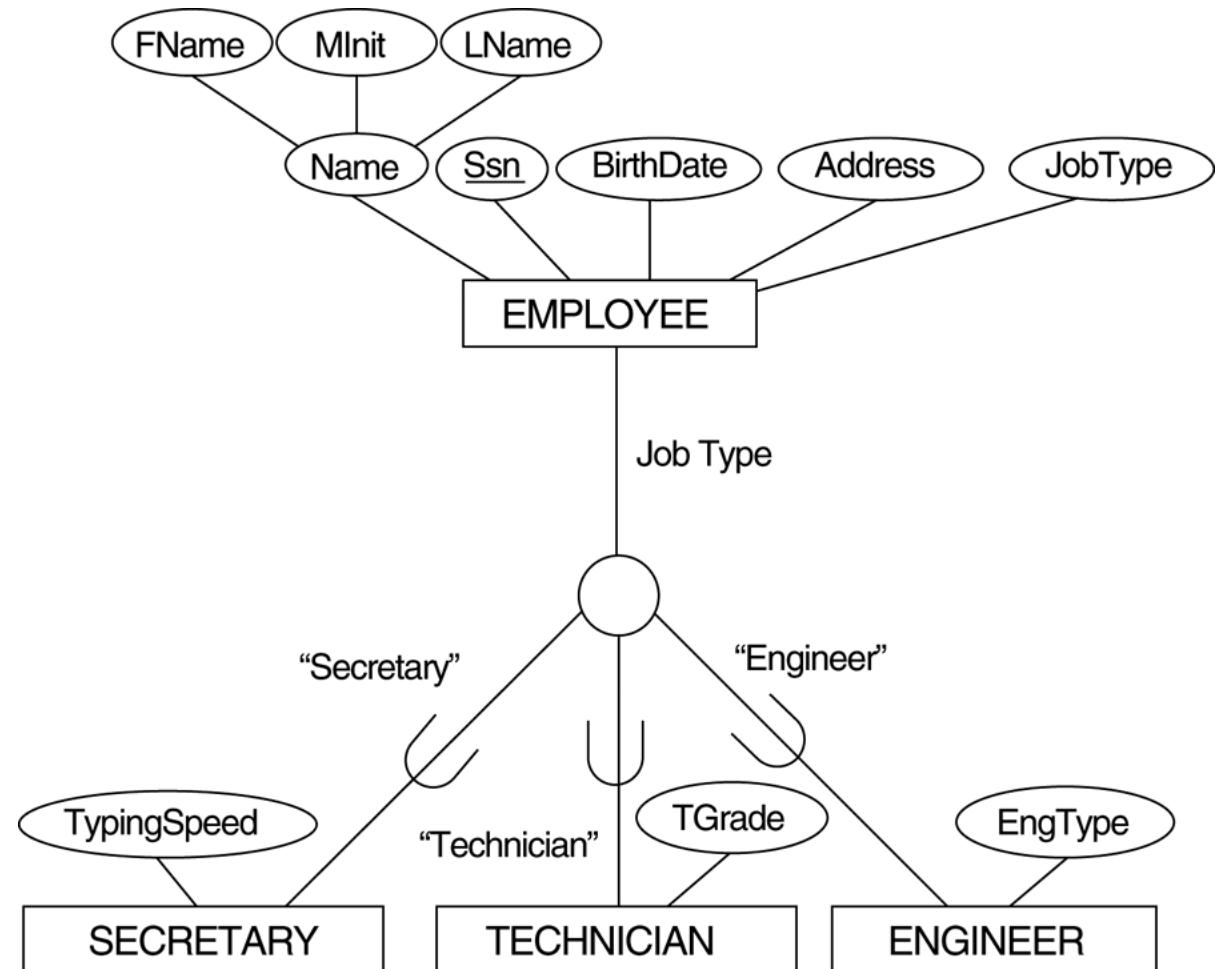
- **Option 8A: Multiple relations-Superclass and subclasses**
  - Create a relation L for C with attributes  $\text{Attrs}(L) = \{k, a_1, \dots, a_n\}$  and  $\text{PK}(L) = k$ . Create a relation  $L_i$  for each subclass  $S_i$ ,  $1 < i < m$ , with the attributes  $\text{Attrs}(L_i) = \{k\} \cup \{\text{attributes of } S_i\}$  and  $\text{PK}(L_i) = k$ . This option works for any specialization (total or partial, disjoint or overlapping).
- **Option 8B: Multiple relations-Subclass relations only**
  - Create a relation  $L_i$  for each subclass  $S_i$ ,  $1 < i < m$ , with the attributes  $\text{Attr}(L_i) = \{\text{attributes of } S_i\} \cup \{k, a_1, \dots, a_n\}$  and  $\text{PK}(L_i) = k$ . This option only works for a specialization whose subclasses are total (every entity in the superclass must belong to (at least) one of the subclasses).

## Mapping EER Model Constructs to Relations (contd.)

- **Option 8C: Single relation with one type attribute**
  - Create a single relation L with attributes  $\text{Attrs}(L) = \{k, a_1, \dots, a_n\} \cup \{\text{attributes of } S_1\} \cup \dots \cup \{\text{attributes of } S_m\} \cup \{t\}$  and  $\text{PK}(L) = k$ . The attribute t is called a type (or **discriminating**) attribute that indicates the subclass to which each tuple belongs
- **Option 8D: Single relation with multiple type attributes**
  - Create a single relation schema L with attributes  $\text{Attrs}(L) = \{k, a_1, \dots, a_n\} \cup \{\text{attributes of } S_1\} \cup \dots \cup \{\text{attributes of } S_m\} \cup \{t_1, t_2, \dots, t_m\}$  and  $\text{PK}(L) = k$ . Each  $t_i$ ,  $1 < i < m$ , is a Boolean type attribute indicating whether a tuple belongs to the subclass  $S_i$ .

## FIGURE 4.4

EER diagram notation for an attribute-defined specialization on JobType.



# Mapping the EER schema in Figure 4.4 (d: Mflag and Pflag)

## (a) EMPLOYEE

<u>SSN</u>	FName	MInit	LName	BirthDate	Address	JobType
------------	-------	-------	-------	-----------	---------	---------

## SECRETARY

<u>SSN</u>	TypingSpeed
------------	-------------

## TECHNICIAN

<u>SSN</u>	TGrade
------------	--------

## ENGINEER

<u>SSN</u>	EngType
------------	---------

## (b) CAR

<u>VehicleId</u>	LicensePlateNo	Price	MaxSpeed	NoOfPassengers
------------------	----------------	-------	----------	----------------

## TRUCK

<u>VehicleId</u>	LicensePlateNo	Price	NoOfAxles	
------------------	----------------	-------	-----------	--

## (c) EMPLOYEE

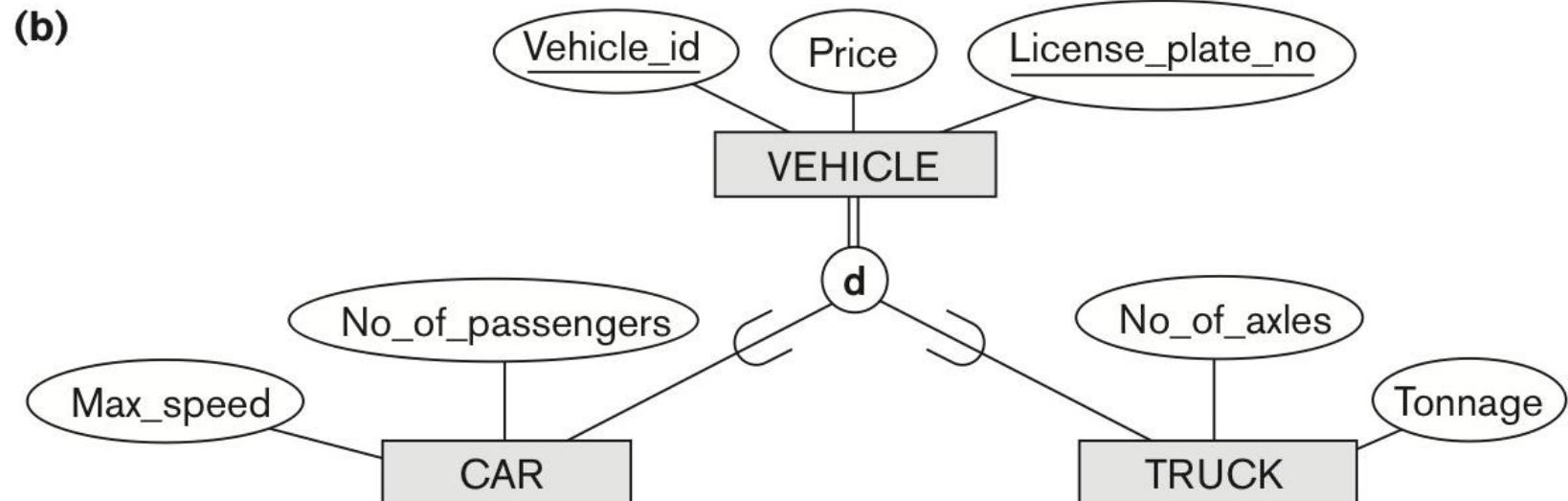
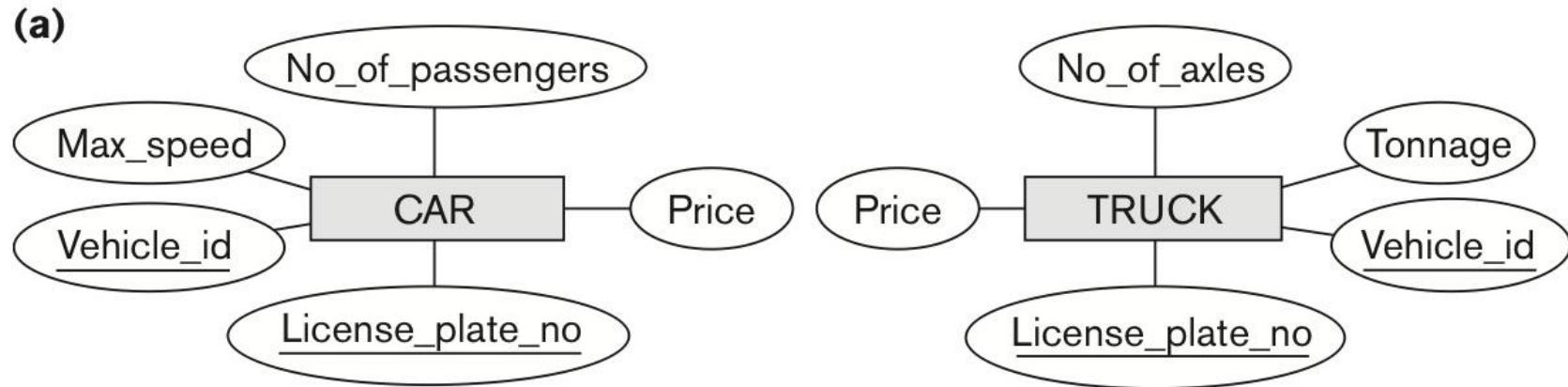
<u>SSN</u>	FName	MInit	LName	BirthDate	Address	JobType	TypingSpeed	TGrade	
------------	-------	-------	-------	-----------	---------	---------	-------------	--------	--

## (d) PART

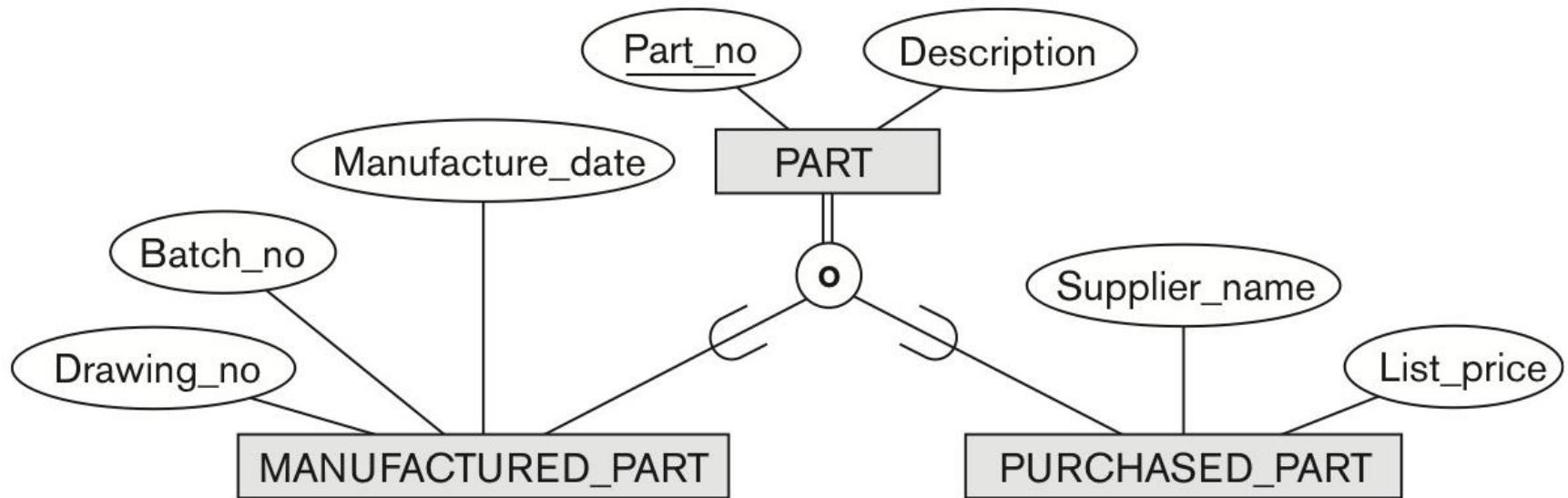
<u>PartNo</u>	Description	MFlag	DrawingNo	ManufactureDate	BatchNo	PFlag	SupplierName	ListPrice
---------------	-------------	-------	-----------	-----------------	---------	-------	--------------	-----------

## FIGURE 4.3 (b)

Generalizing CAR and TRUCK into the superclass VEHICLE.

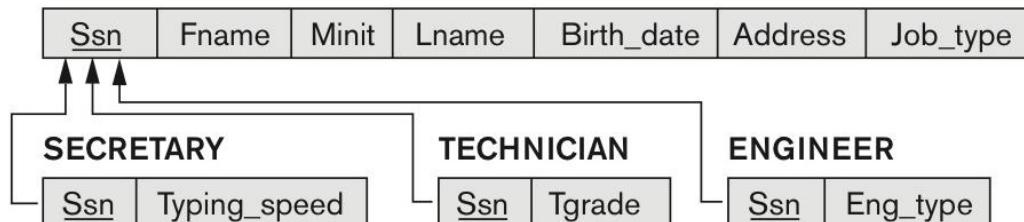


**FIGURE 4.5**  
An overlapping (non-disjoint) specialization.



# Fig. 9.5: Different Options for Mapping Generalization Hierarchies - summary

## (a) EMPLOYEE



## (b) CAR

<u>Vehicle_id</u>	<u>License_plate_no</u>	<u>Price</u>	<u>Max_speed</u>	<u>No_of_passengers</u>
-------------------	-------------------------	--------------	------------------	-------------------------

### TRUCK

<u>Vehicle_id</u>	<u>License_plate_no</u>	<u>Price</u>	<u>No_of_axles</u>	<u>Tonnage</u>
-------------------	-------------------------	--------------	--------------------	----------------

## (c) EMPLOYEE

<u>Ssn</u>	<u>Fname</u>	<u>Minit</u>	<u>Lname</u>	<u>Birth_date</u>	<u>Address</u>	<u>Job_type</u>	<u>Typing_speed</u>	<u>Tgrade</u>	<u>Eng_type</u>
------------	--------------	--------------	--------------	-------------------	----------------	-----------------	---------------------	---------------	-----------------

## (d) PART

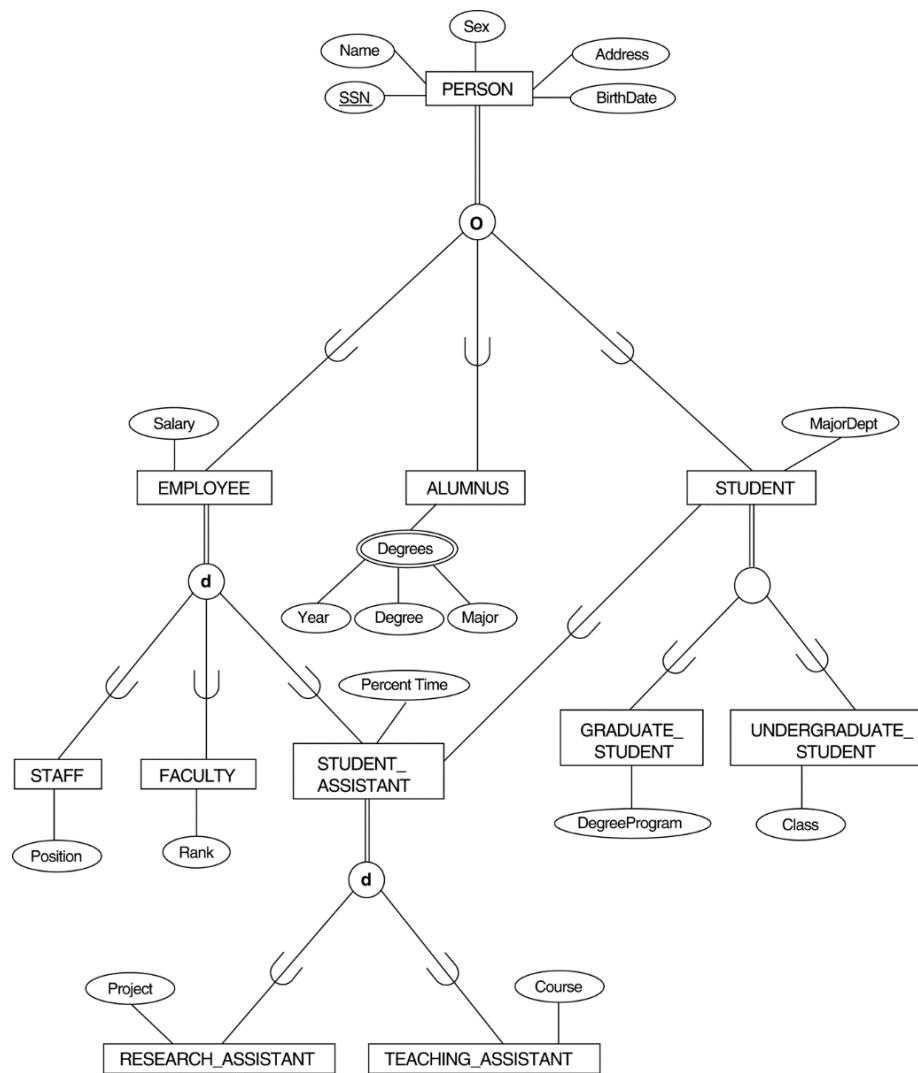
<u>Part_no</u>	<u>Description</u>	<u>Mflag</u>	<u>Drawing_no</u>	<u>Manufacture_date</u>	<u>Batch_no</u>	<u>Pflag</u>	<u>Supplier_name</u>	<u>List_price</u>
----------------	--------------------	--------------	-------------------	-------------------------	-----------------	--------------	----------------------	-------------------

# Mapping EER Model Constructs to Relations (contd.)

- Mapping of Shared Subclasses (Multiple Inheritance)
  - A shared subclass, such as STUDENT\_ASSISTANT, is a subclass of several classes, indicating multiple inheritance. These classes must all have the same key attribute; otherwise, the shared subclass would be modeled as a category.
  - We can apply any of the options discussed in Step 8 to a shared subclass, subject to the restriction discussed in Step 8 of the mapping algorithm. Below both 8C and 8D are used for the shared class STUDENT\_ASSISTANT.

**FIGURE 4.7**

A specialization lattice with multiple inheritance for a UNIVERSITY database.



**FIGURE 9.6**  
Mapping the EER specialization lattice in Figure 4.7 using multiple options.

**PERSON**

<u>Ssn</u>	Name	Birth_date	Sex	Address
------------	------	------------	-----	---------

**EMPLOYEE**

<u>Ssn</u>	Salary	Employee_type	Position	Rank	Percent_time	Ra_flag	Ta_flag	Project	Course
------------	--------	---------------	----------	------	--------------	---------	---------	---------	--------

**ALUMNUS**

<u>Ssn</u>
------------

**ALUMNUS\_DEGREES**

<u>Ssn</u>	Year	Degree	Major
------------	------	--------	-------

**STUDENT**

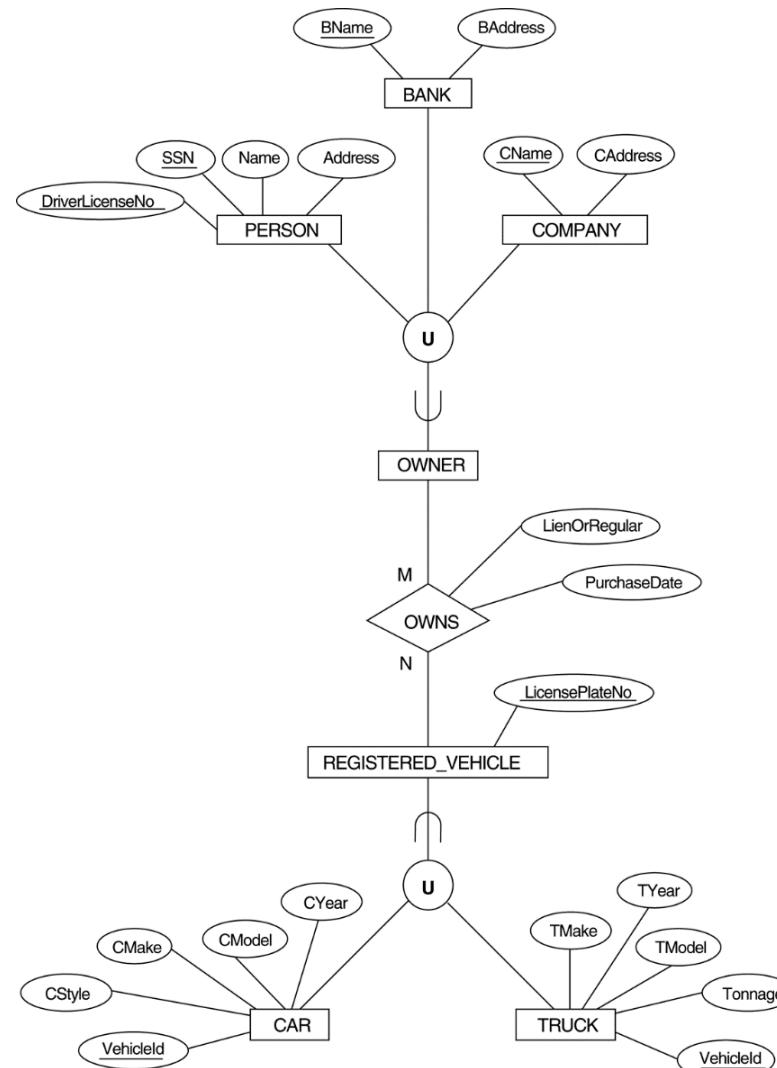
<u>Ssn</u>	Major_dept	Grad_flag	Undergrad_flag	Degree_program	Class	Student_assist_flag
------------	------------	-----------	----------------	----------------	-------	---------------------

# Mapping EER Model Constructs to Relations (contd.)

- **Step 9: Mapping of Union Types (Categories).**
  - For mapping a category whose defining superclass have different keys, it is customary to specify a new key attribute, called a surrogate key, when creating a relation to correspond to the category.
  - In the example below we can create a relation OWNER to correspond to the OWNER category and include any attributes of the category in this relation. The primary key of the OWNER relation is the surrogate key, which we called OwnerId.

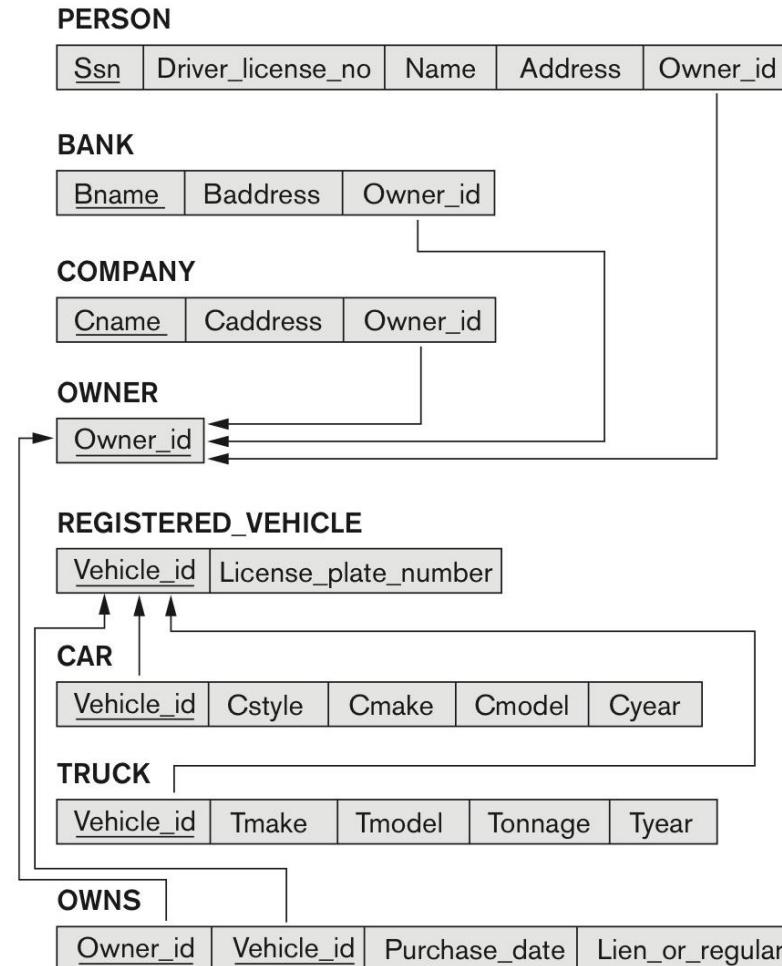
FIGURE 4.8

Two categories (union types): OWNER and REGISTERED\_VEHICLE.



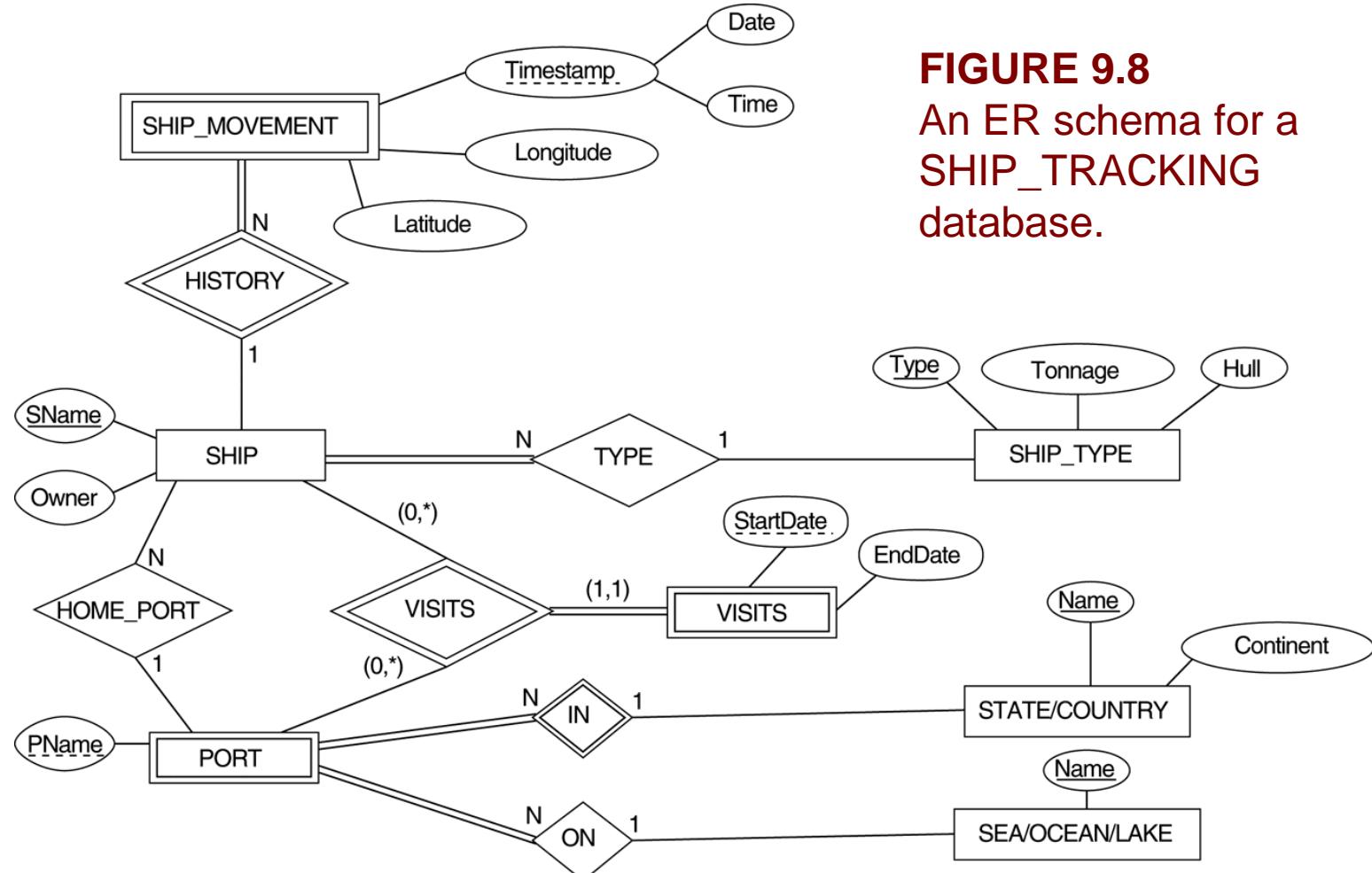
## FIGURE 9.7

### Mapping the EER categories (union types) in Figure 4.8 to relations.



# Mapping Exercise-1

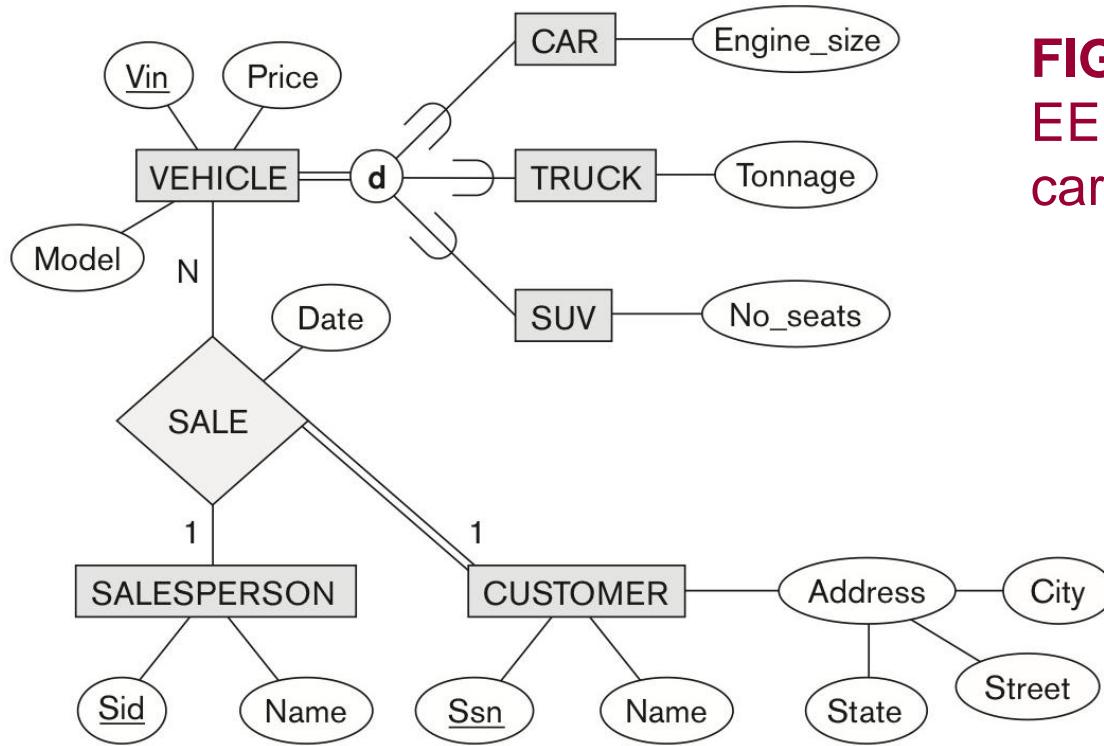
Exercise 9.4 : Map this schema into a set of relations.



**FIGURE 9.8**  
An ER schema for a  
SHIP\_TRACKING  
database.

# Mapping Exercise-2

Exercise 9.9 : Map this schema into a set of relations



**FIGURE 9.9**  
EER diagram for a  
car dealer



# Part 2: Relational Databases

Kulsawasd Jitkajornwanich, PhD

kulsawasd.ji@kmitl.ac.th

[Taken and adapted from slides and/or other materials by Ramez Elmasri and Shamkant B. Navathe,  
“Fundamentals of Database Systems” (7<sup>th</sup> Edition), Addison-Wesley, Pearson]



# Chapter 6: Disk Storage and File Structures

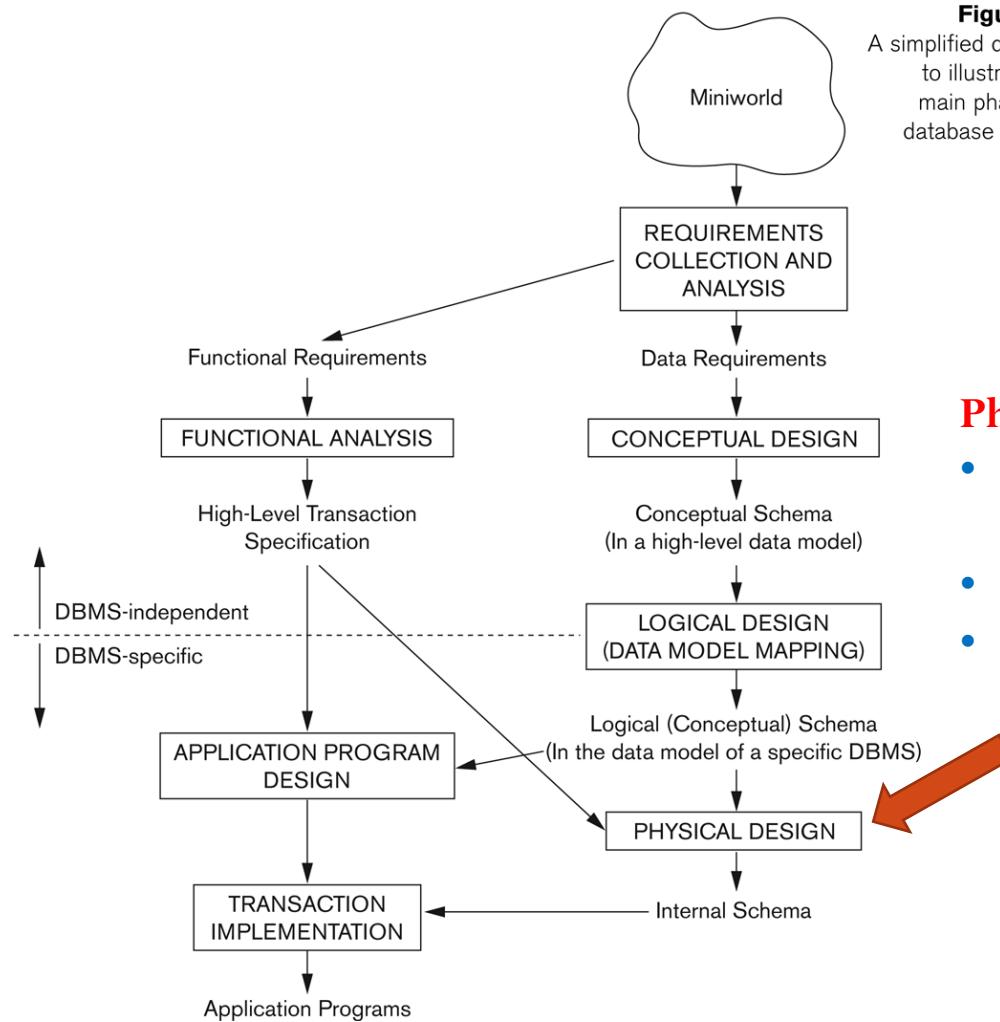


คณะวิทยาศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง  
Faculty of Science, King Mongkut's Institute of Technology Ladkrabang

# Outline

- Disk Storage Devices
- Files of Records
- Operations on Files
- Unordered Files
- Ordered Files
- Hashed Files

# Overview of Database Design Process



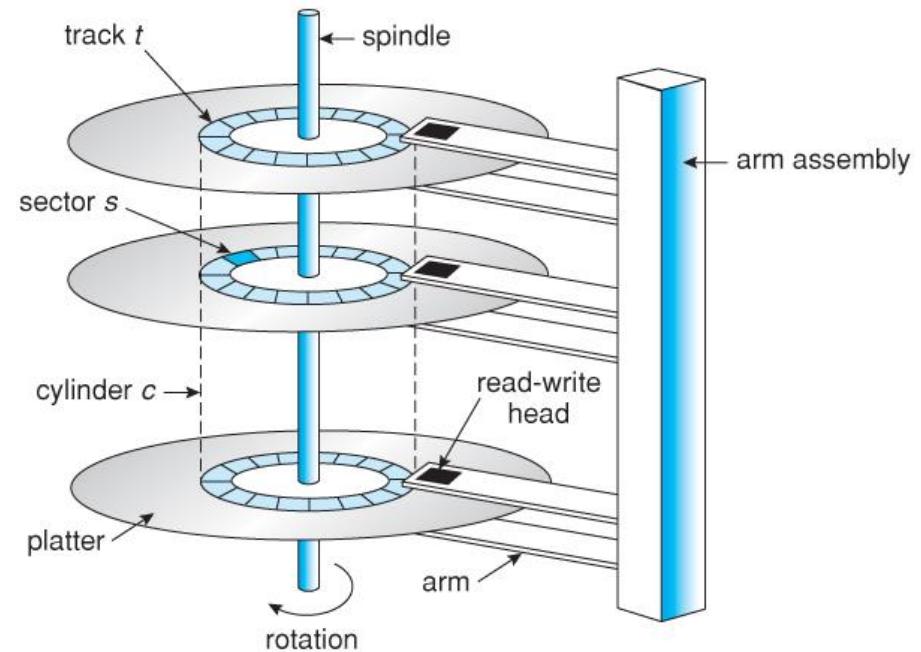
**Figure 3.1**  
A simplified diagram  
to illustrate the  
main phases of  
database design.

## Physical design:

- takes into account 1) logical mapped tables and 2) functional requirements
- may need to re-consider denormalization
- requires basic understanding of storage structures & indexing

# Disk Storage Devices

- Relational data are physically stored in a *secondary storage device* for “high” storage capacity and “low” cost.
- How do they look like?



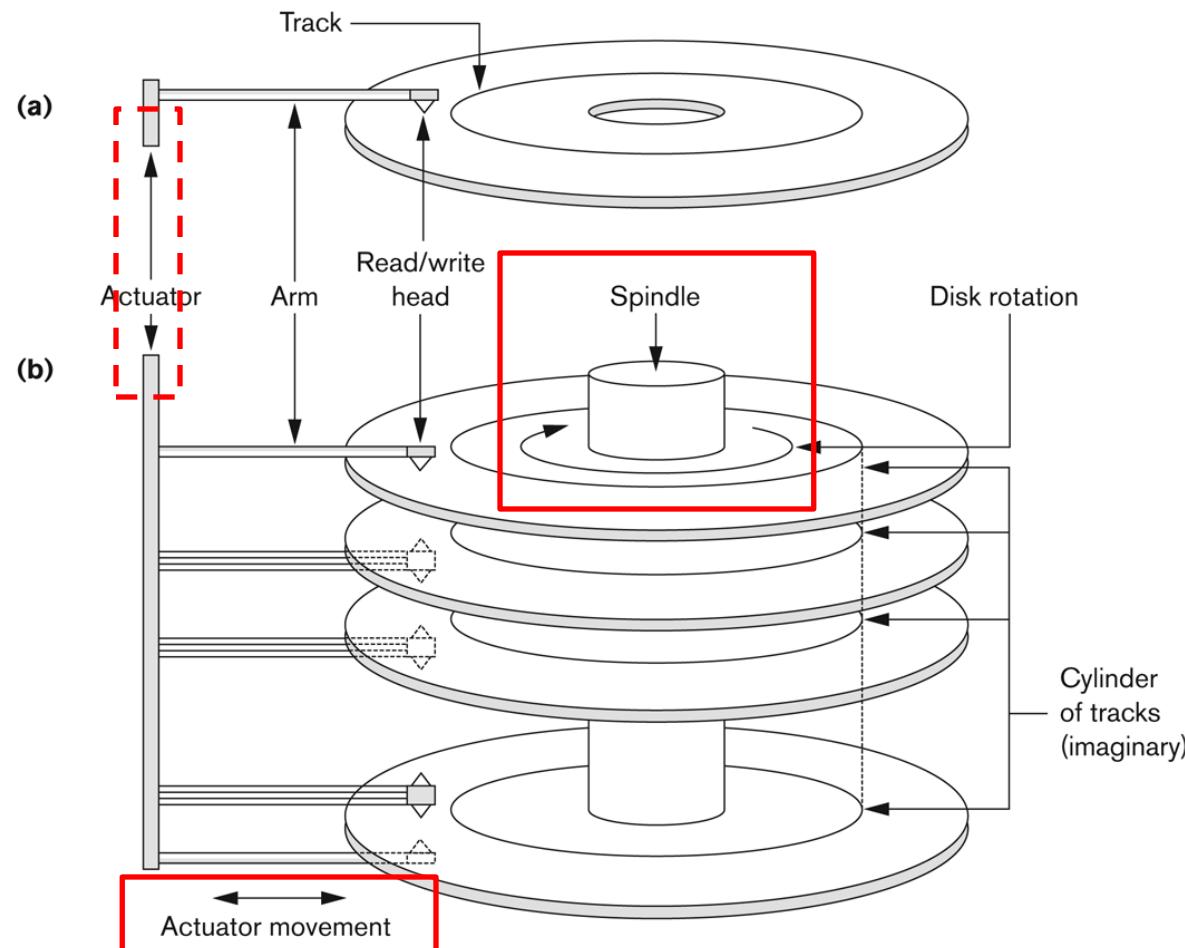
# Disk Storage Devices

- Relational data are physically stored in a *secondary storage device* for “high” storage capacity and “low” cost.
- How do they look like?
  - Data stored as magnetized areas on magnetic disk surfaces as **bit** (0 or 1).
  - (8) bits are grouped together into **byte** to represent a *character*
  - A **disk pack** contains several magnetic disks connected to a rotating spindle.
  - Disks are divided into concentric circular **tracks** on each disk **surface**.
    - Track capacities vary typically from 4 to 50 Kbytes or more

# Disk Storage Devices (contd.)

**Figure 13.1**

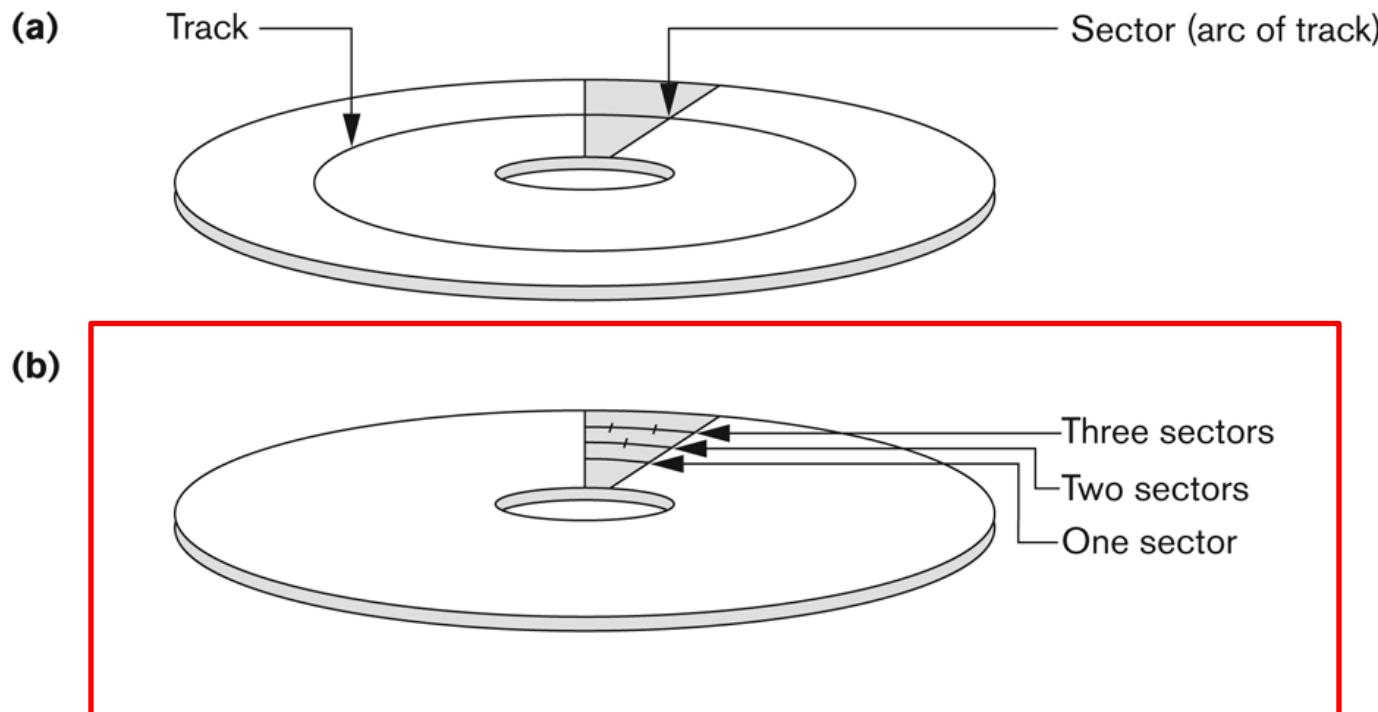
(a) A single-sided disk with read/write hardware. (b) A disk pack with read/write hardware.



# Disk Storage Devices (contd.)

- A track is divided into smaller **blocks** or **sectors**
  - because it usually contains a large amount of information
- The division of a track into **sectors/blocks** is hard-coded on the disk surface and CANNOT be changed.
- A track is divided into **blocks**.
  - The block size  $B$  is “fixed” for each system.
    - Typical block sizes range from  $B=512$  bytes to  $B=4096$  bytes.
    - *Whole blocks* are transferred between disk and main memory for processing.

# Disk Storage Devices (contd.)



**Figure 13.2**  
Different sector organizations on disk.  
(a) Sectors subtending a fixed angle.  
(b) Sectors maintaining a uniform recording density.

# Disk Storage Devices (contd.)

- A **read-write head** moves to the “track” that contains the block to be transferred.
  - Disk rotation moves the “block” under the read-write head for reading or writing.
- A physical disk block (hardware) address consists of:
  - a *cylinder number* (imaginary collection of tracks of same radius from all recorded surfaces)
  - the *track number* or *surface number* (within the cylinder)
  - and *block number* (within track).
- Reading or writing a disk block is time consuming because of the seek time *s* and rotational delay (latency) *rd*.

# Typical Disk Parameters

**Table 13.1**  
Specifications of Typical High-end Cheetah Disks from Seagate

Description	Cheetah 10K.6	Cheetah 10K.7
Model Number	ST3146807LC	ST3300007LW
Form Factor (width)	3.5 inch	3.5 inch
Form Factor (height)	1 inch	1 inch
Height	25.4 mm	25.4 mm
Width	101.6 mm	101.6 mm
Length	146.05 mm	146.05 mm
Weight	0.73 Kg	0.726 kg
Capacity/Interface		
Formatted Capacity	146.8 Gbytes	300 Gbytes
Interface Type	80-pin	68-pin
Configuration		
Number of disks (physical)	4	4
Number of heads (physical)	8	8
Number of Cylinders	49,854	90,774
Total Tracks		726,192
Bytes per Sector	512	512
Areal Density	36,000 Mb/sq.inch	
Track Density	64,000 Tracks/inch	105,000 Tracks/inch
Recording Density	570,000 bits/inch	658,000 bits/inch
Bytes/Track (avg)		556
Performance		
Transfer Rates		
Internal Transfer Rate (min)	475 Mb/sec	472 Mb/sec
Internal Transfer Rate (max)	840 Mb/sec	944 Mb/sec
Formated Int. Transfer Rate (min)	43 MB/sec	59 MB/sec
Formated Int. Transfer Rate (max)	78 MB/sec	118 MB/sec
External I/O Transfer Rate (max)	320 MB/sec	320 MB/sec
Average Formatted Transfer Rate	59.9 MB/sec	59.5 MB/sec
Seek Times		
Avg. Seek Time (Read)	4.7 ms (typical)	4.7 ms (typical)
Avg. Seek Time (Write)	5.2 ms (typical)	5.3 ms (typical)
Track-to-track, Seek, Read	0.3 ms (typical)	0.2 ms (typical)
Track-to-track, Seek, Write	0.5 ms (typical)	0.5 ms (typical)
Full Disc Seek, Read		9.5 ms (typical)
Full Disc Seek, Write		10.3 ms (typical)
Average Latency	2.99 ms	3 msec
Other		
Default Buffer (cache) size	8,000 KB	8,192 KB
Spindle Speed	10000 RPM	10000 RPM
Power-on to Ready Time		25 sec

(Courtesy of Seagate Technology)

# Records

- **Fixed and variable length** records
- Records contain fields which have values of a particular type
  - E.g., amount, date, time, age
- Fields themselves may be fixed length or variable length
- Variable length fields can be mixed into one record:
  - *Separator characters* or length fields are needed so that the record can be “parsed.”

# Blocking

- **Blocking:**
  - Refers to “storing” a number of records in one block on the disk.
- Blocking factor (**bfr**) refers to “the number of records” per block.
- There may be empty space in a block if an integral number of records do not fit in one block.
- **Spanned Records:**
  - Refers to records that exceed the size of one or more blocks and hence span a number of blocks.

# Files of Records

- A **file** is a “*sequence*” of records, where each record is a collection of data values (or data items).
- A **file descriptor** (or **file header**) includes information that describes the file, such as the *field names* and their *data types*, and the addresses of the file blocks on disk.
- Records are stored on disk blocks.
- The **blocking factor bfr** for a file is the (average) number of file records stored in a disk block.
- A file can have **fixed-length** records or **variable-length** records.

# Files of Records (contd.)

- File records can be **unspanned** or **spanned**
  - **Unspanned**: no record can span two blocks
  - **Spanned**: a record can be stored in more than one block
- The physical disk blocks that are allocated to hold the records of a file can be *contiguous, linked, or indexed*.
- In a file of *fixed-length records*, all records have the same format. Usually, *unspanned blocking* is used with such files.
- Files of *variable-length records* require additional information to be stored in each record, such as **separator characters** and **field types**.
  - Usually *spanned blocking* is used with such files.

# Operation on Files

- Typical file operations include:
  - **OPEN:** Readies the file for access, and associates a pointer that will refer to a *current* file record at each point in time.
  - **FIND:** Searches for the first file record that satisfies a certain condition, and makes it the current file record.
  - **FINDNEXT:** Searches for the next file record (from the current record) that satisfies a certain condition, and makes it the current file record.
  - **READ:** Reads the current file record into a program variable.
  - **INSERT:** Inserts a new record into the file & makes it the current file record.
  - **DELETE:** Removes the current file record from the file, usually by marking the record to indicate that it is no longer valid.
  - **MODIFY:** Changes the values of some fields of the current file record.
  - **CLOSE:** Terminates access to the file.
  - **REORGANIZE:** Reorganizes the file records.
    - For example, the records marked deleted are physically removed from the file or a new organization of the file records is created.
  - **READ\_ORDERED:** Read the file blocks in order of a specific field of the file.

# Unordered Files

- Also called a **heap** or a **pile** file.
- New records are inserted at the end of the file.
- A **linear search** through the file records is necessary to *search for a record*.
  - This requires reading and **searching** half the file blocks on the average, and is hence quite expensive.
- *Record insertion* is quite efficient.
- *Reading the records in order* of a particular field requires sorting the file records.

# Ordered Files

- Also called a **sequential** file.
- 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* (or *transaction*) 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*  $\log_2$  of the file blocks on the average, an improvement over linear search.
  - *Reading the records in order* of the ordering field is quite efficient.

# Ordered Files (contd.)

	NAME	SSN	BIRTHDATE	JOB	SALARY	SEX
block 1	Aaron, Ed					
	Abbott, Diane					
						⋮
	Acosta, Marc					
block 2	Adams, John					
	Adams, Robin					
						⋮
	Akers, Jan					
block 3	Alexander, Ed					
	Alfred, Bob					
						⋮
	Allen, Sam					
block 4	Allen, Troy					
	Anders, Keith					
						⋮
	Anderson, Rob					
block 5	Anderson, Zach					
	Angeli, Joe					
						⋮
	Archer, Sue					
block 6	Arnold, Mack					
	Arnold, Steven					
						⋮
	Atkins, Timothy					
						⋮
block n -1	Wong, James					
	Wood, Donald					
						⋮
	Woods, Manny					
block n	Wright, Pam					
	Wyatt, Charles					
						⋮
	Zimmer, Byron					

# Average Access Times

- The following table shows the average access time to access a specific record for a given type of file

**TABLE 13.2 AVERAGE ACCESS TIMES FOR BASIC FILE ORGANIZATIONS**

TYPE OF ORGANIZATION	ACCESS/SEARCH METHOD	AVERAGE TIME TO ACCESS A SPECIFIC RECORD
Heap (Unordered)	Sequential scan (Linear Search)	$b/2$
Ordered	Sequential scan	$b/2$
Ordered	Binary Search	$\log_2 b$

# Hashed Files

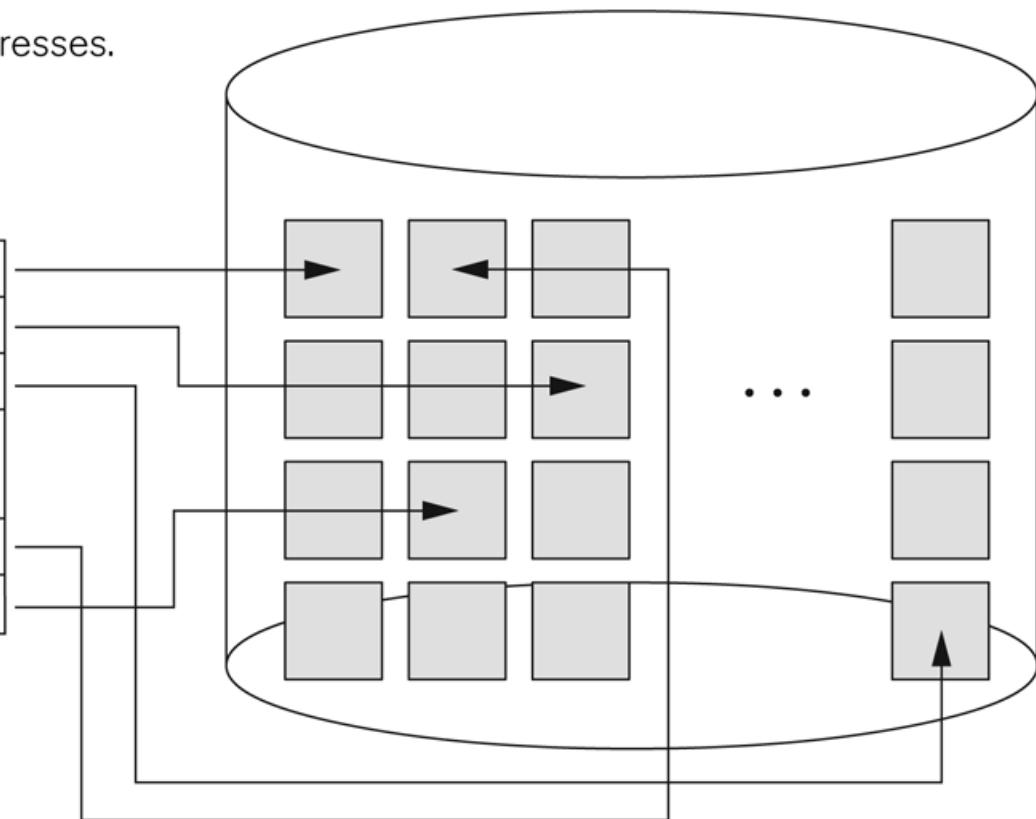
- Hashing for disk files is called **External Hashing**
- The file blocks are divided into  $M$  equal-sized **buckets**, numbered  $\text{bucket}_0, \text{bucket}_1, \dots, \text{bucket}_{M-1}$ .
  - Typically, a bucket corresponds to *one (or a fixed number of) disk block*.
- One of the file fields is designated to be the **hash key** of the file.
- The record with hash key value  $K$  is stored in bucket  $i$ , where  $i=h(K)$ , and  $h$  is the **hashing function**.
- *Search* is very efficient on the hash key.
- “Collisions” occur when a new record hashes to a bucket that is *already full*.
  - An overflow file is kept for storing such records.
  - Overflow records that hash to each bucket can be linked together.

# Hashed Files (contd.)

**Figure 13.9**

Matching bucket numbers to disk block addresses.

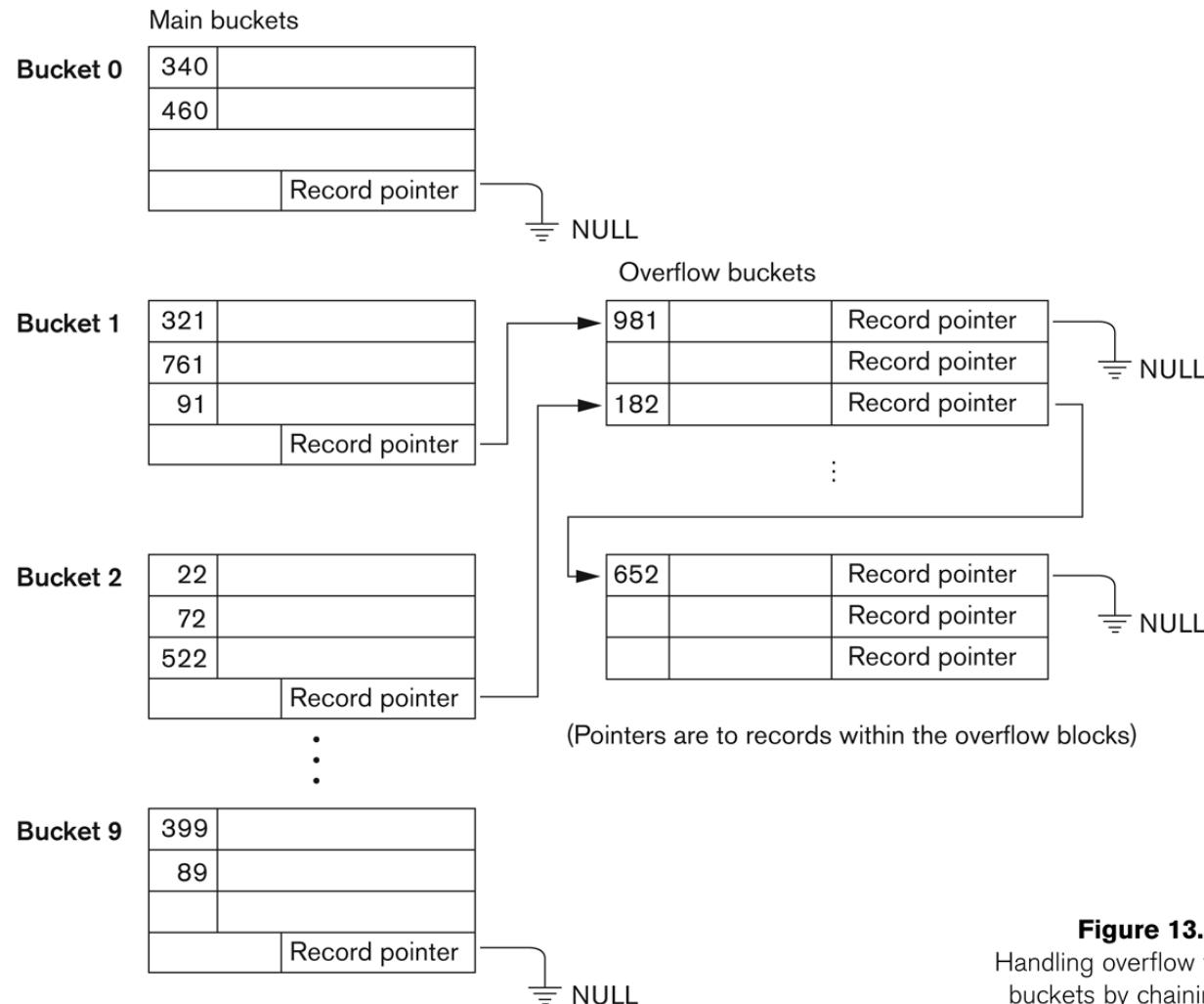
Bucket Number	Block address on disk
0	
1	
2	
$\vdots$	
$M - 2$	
$M - 1$	



# Hashed Files (contd.)

- There are numerous methods for collision resolution, including the following:
  - **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.
  - **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 record location. A collision is resolved by placing the new record in an unused overflow location and setting the pointer of the occupied hash address location to the address of that overflow location.
  - **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.

# Hashed Files - Overflow handling



**Figure 13.10**  
Handling overflow for  
buckets by chaining.

# Hashed Files (contd.)

- To reduce overflow records, a hash file is typically kept 70-80% full.
- The hash function  $h$  should distribute the records uniformly among the buckets
  - Otherwise, search time will be increased because many overflow records will exist.
- Main disadvantages of static external hashing:
  - *Fixed number of buckets*  $M$  is a problem if the number of records in the file grows or shrinks.
  - *Ordered access* on the hash key is quite inefficient (requires sorting the records).
  - What about *insertion* performance?

# Chapter 7: Indexing Structures



คณะวิทยาศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง  
Faculty of Science, King Mongkut's Institute of Technology Ladkrabang

# Outline

- Types of Single-level Ordered Indexes
  - Primary Indexes
  - Clustering Indexes
  - Secondary Indexes
- Factors that influence physical database design
- Physical database design decisions

# Indexes as Access Paths

- A single-level index is an auxiliary file that makes it more efficient to search for a record in the data file.
- The index is usually specified on one field of the file (although it could be specified on several fields)
- *One form* of an index is a file of entries **<field value, pointer to record>**, which is *ordered by field value*
- The index is called an *access path* on the field.

# Indexes as Access Paths (contd.)

- The index file usually occupies *considerably less disk blocks* than the data file because its entries are much smaller
- A “binary search” on the index *yields* a pointer to the file record
- Indexes can also be characterized as dense or sparse
  - A **dense index** has an index entry for every search key value (and hence every record) in the data file.
  - A **sparse (or nondense) index**, on the other hand, has index entries for only some of the search values

# Indexes as Access Paths (contd.)

- **Example:** Given the following data file EMPLOYEE(NAME, SSN, ADDRESS, JOB, SAL, ... )
- Suppose that:
  - record size  $R=150$  bytes, block size  $B=512$  bytes,  $r=30,000$  records
- Then, we get:
  - blocking factor  $Bfr=B \text{ div } R=512 \text{ div } 150=3$  records/block
  - number of file blocks  $b=(r/Bfr)=(30,000/3)=10,000$  blocks
- For an index on the SSN field, assume the field size  $V_{SSN}=9$  bytes, assume the record pointer size  $P_R=7$  bytes. Then:
  - index entry size  $R_I=(V_{SSN}+P_R)=(9+7)=16$  bytes
  - index blocking factor  $Bfr_I=B \text{ div } R_I=512 \text{ div } 16=32$  entries/block
  - number of index blocks  $b=(r/Bfr_I)=(10,000/32)=313$  blocks
  - binary search needs  $\log_2 bI=\log_2 313=9$  block accesses
  - This is compared to an average linear search cost of:
    - $(b/2)=10,000/2=5,000$  block accesses
  - If the file records are ordered, the binary search cost would be:
    - $\log_2 b=\log_2 10,000=14$  block accesses

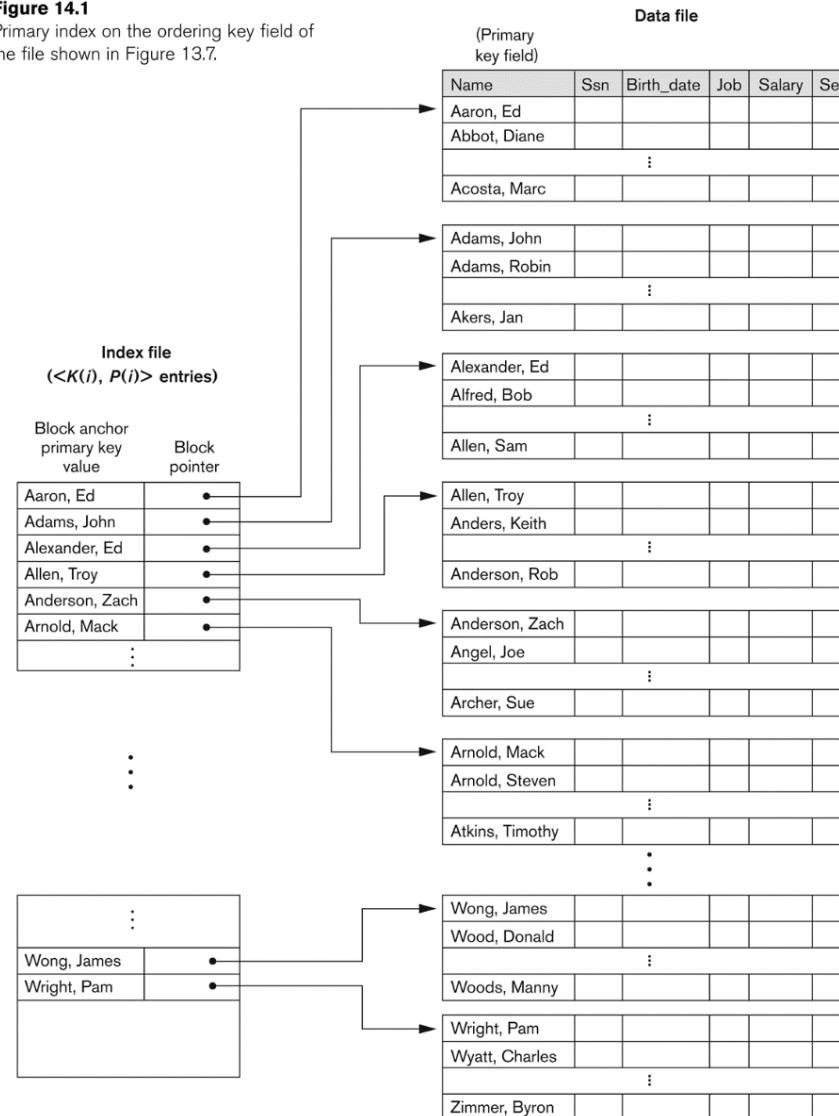
# Types of Single-Level Indexes

- Primary Index
  - Defined on an *ordered data file*
  - The data file is ordered on a **key field**
  - Includes one index entry *for each block* in the data file; the index entry has the key field value for the *first record* in the block, which is called the *block anchor*
  - A similar scheme can use the *last record* in a block.
  - A primary index is a nondense (*sparse*) index, since it includes an entry for each disk block of the data file and the keys of its anchor record rather than for every search value.

# Primary index on the ordering key field

Figure 14.1

Primary index on the ordering key field of the file shown in Figure 13.7.

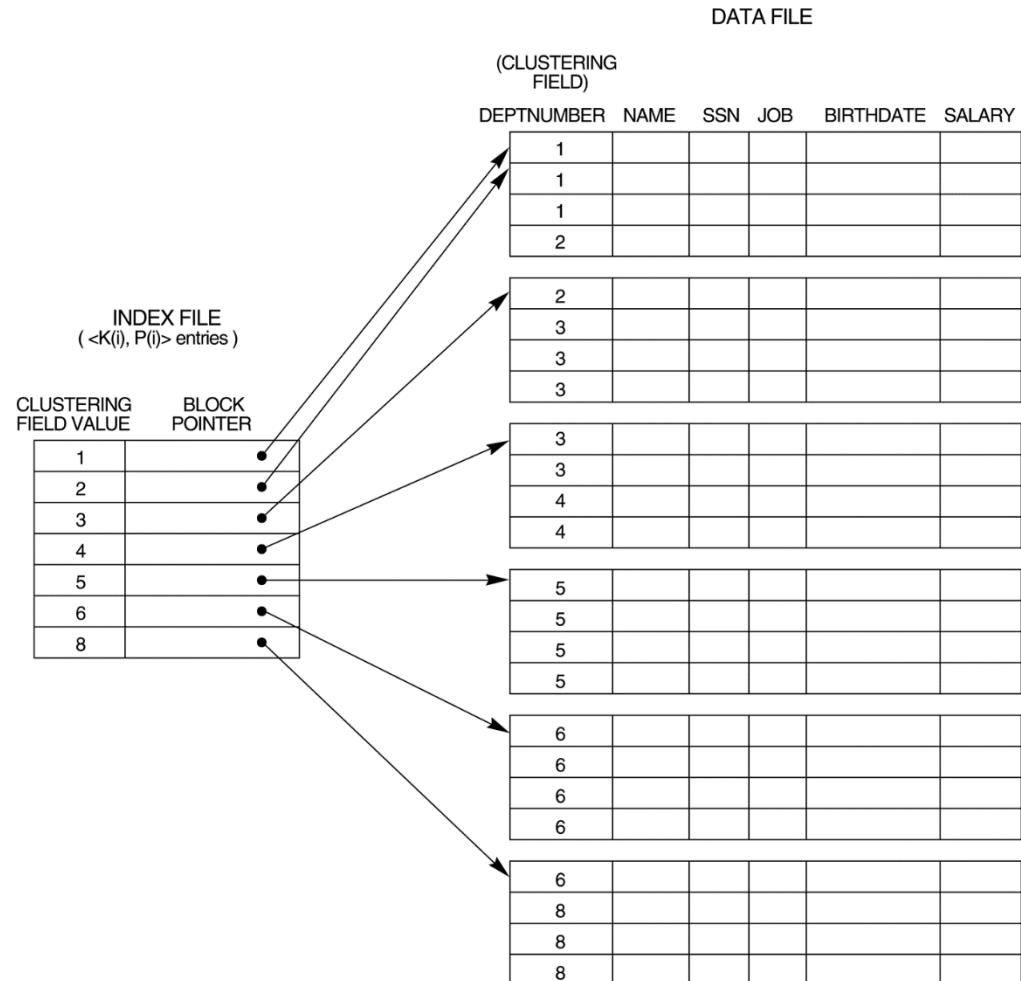


# Types of Single-Level Indexes

- Clustering Index
  - Defined on an *ordered data file*
  - The data file is ordered on a *non-key field* unlike primary index, which requires that the ordering field of the data file have a distinct value for each record.
  - Includes one index entry *for each distinct value* of the field; the index entry points to the first data block that contains records with that field value.
  - It is another example of *nondense* index where Insertion and Deletion is relatively straightforward with a clustering index.

# A Clustering Index Example

- FIGURE 14.2  
A clustering index on  
the DEPTNUMBER  
ordering non-key  
field of an  
EMPLOYEE file.



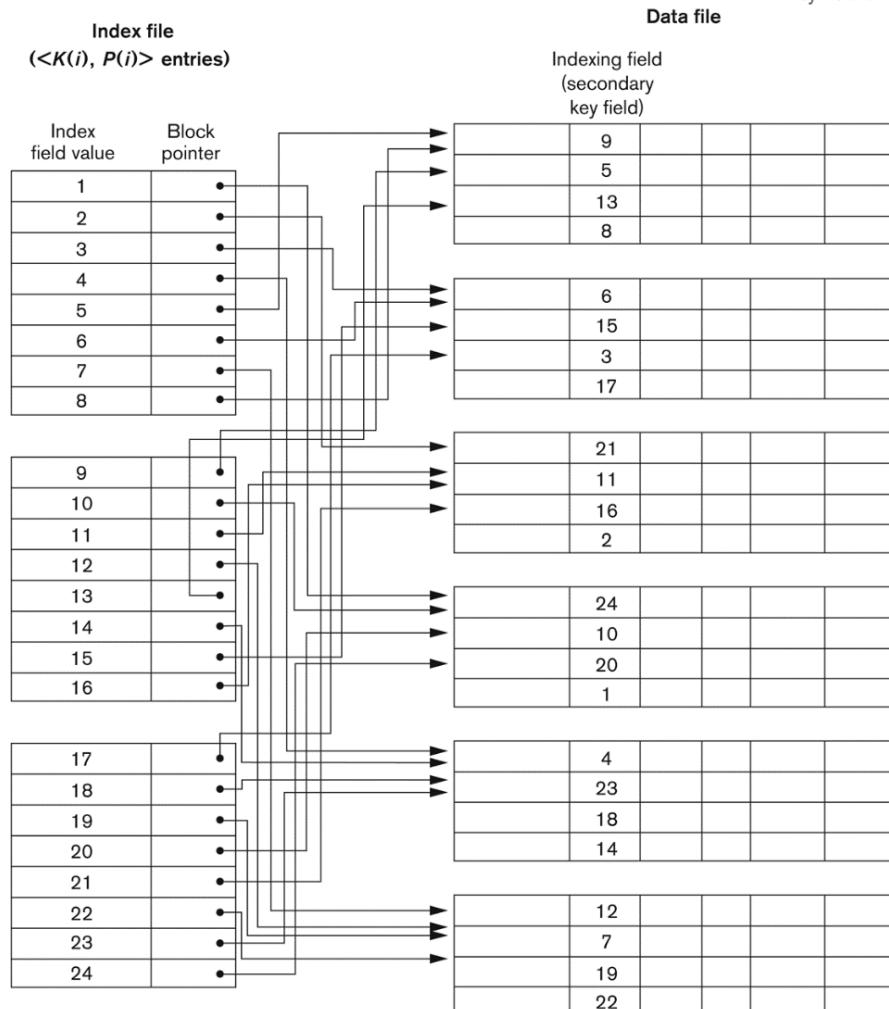
# Types of Single-Level Indexes

- Secondary Index
  - A secondary index provides a *secondary means* of accessing a file for which *some primary access already exists*.
  - The secondary index may be on a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values.
  - The index is an ordered file with two fields.
    - The first field is of the same data type as some **non-ordering field** of the data file that is an indexing field.
    - The second field is either a **block** pointer or a record pointer.
    - There can be *many* secondary indexes (and hence, indexing fields) for the same file.
  - Includes one entry *for each record* in the data file; hence, it is a *dense index*

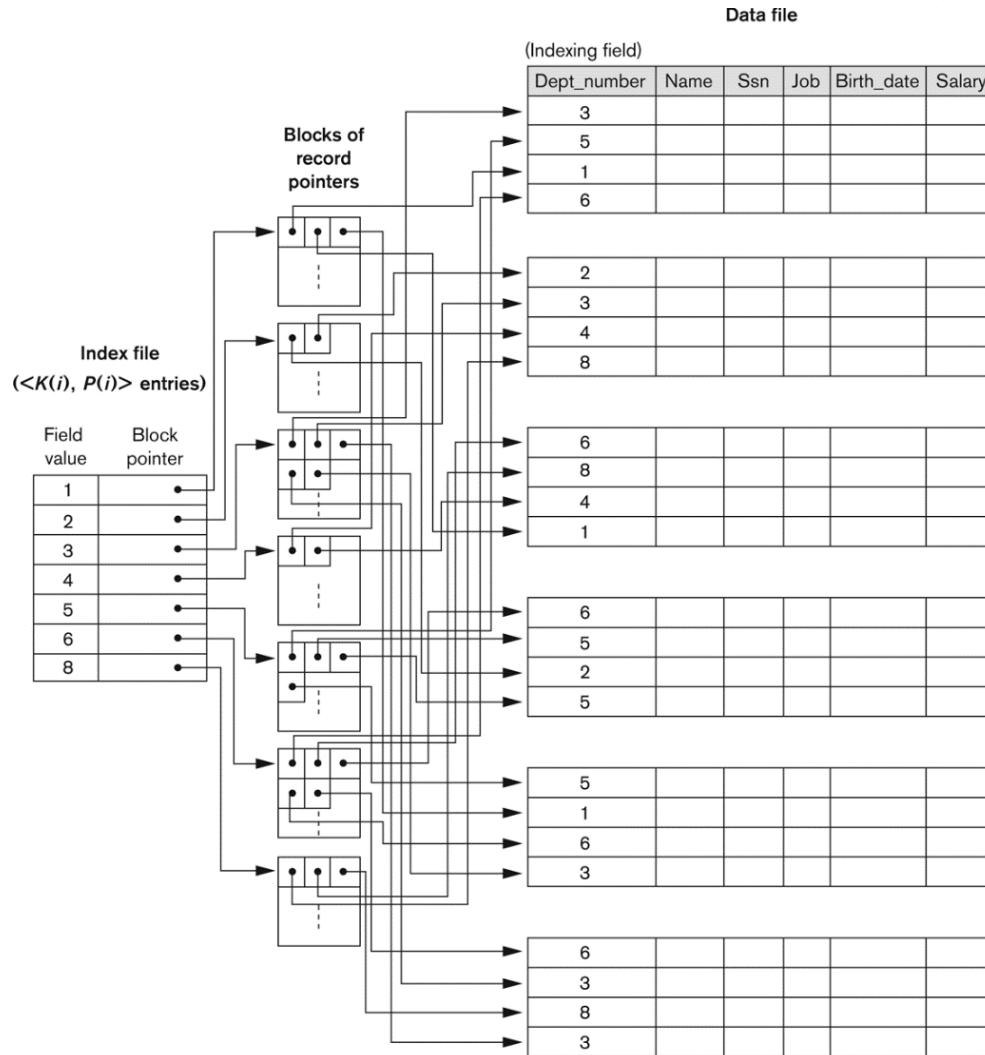
# Example of a Dense Secondary Index

Figure 14.4

A dense secondary index (with block pointers) on a nonordering key field of a file.



# An Example of a Secondary Index



**Figure 14.5**

A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

# Factors that influence physical database design

- There is no *standard guideline* to do physical database design
  - It all depends on “Job mix”, a combination of:
    - Queries, transactions, and applications that run on the database

## A. Analyzing the database *queries* and *transactions*

- Queries:
  1. Files (relations) that will be accessed by the query
  2. *Selection condition* attributes
  3. Whether the selection condition is equality, inequality or range condition
  4. *Join condition* attributes
  5. The attributes whose values will be retrieved by the query
  - Items (2) and (4) are candidates for access structures (indexes, hash keys, sorting of the file)

# Factors that influence physical database design

- There is no *standard guideline* to do physical database design
  - It all depends on “Job mix”, a combination of:
    - Queries, transactions, and applications that run on the database

## A. Analyzing the database *queries* and *transactions*

- Update transactions:
  1. The files that will be updated
  2. Type of operations on each file (insert, update, or delete)
  3. The attributes specified in the selection condition of the operation
  4. The attributes whose values will be changed
  - Item 3's attributes are the candidates for access structures
  - Item 4's attributes are the candidates for *avoiding* an access structure

# Factors that influence physical database design

- B. Analyzing the *expected frequency* of invocation of queries and transactions
  - Pay *extra* attentions to those queries (and transactions) (20%) that are most invoked (80%) of the time (according to informal *80-20 rule*) and analyze their respective statistics (as investigating statistics of all queries/trans. are impossible)
- C. Analyzing the *time constraints* of queries and transactions
  - Queries with responding time constraints are prioritized; the associated attributes are then candidates for primary access structures

# Factors that influence physical database design

- D. Analyzing the *expected frequencies of update* operations
  - The files that are frequently updated should have minimum numbers of indexes.
- E. Analyzing the *uniqueness constraints* on attributes
  - All *candidate keys* should have indexes.

# Physical database design decisions

- Access path options: (1) file organization and (2) indexes (single or composite)
- At most, one of the indexes on each file can be a primary or clustering index
- Any number of additional secondary indexes can be created
- The attributes whose values are specified in *selection* and *join* conditions should have indexes
- Update operations (insert, update, and delete) on the files that have indexes will be significantly slow

# Physical database design decisions

## 1. Whether to index an attribute

- An attribute that is a *key* (unique) or that is specified by the query (selection or join) should be indexed.
- Multiple indexes help speeding up the retrieval because some operations can be achieved by simply scanning the indexes.

## 2. What attribute OR attributes to index on

- A *composite index* is advised if the associated attributes are often queried together
- Note that the *ordering of attributes* matter

# Physical database design decisions

## 3. Whether to set up a *clustered index*

- In RDBMSs, clustered index (primary or clustering index) can be created by specifying keyword “CLUSTERED”
- *At most*, one index per table can be primary or clustering index as the file can “physically” be ordered on ONE (or a set) of attribute(s)
  - If the attribute is key, a primary index is created
  - If the attribute is NOT a key, a clustering index is created
- Range queries benefit greatly from clustered index as the file be physically ordered based on that attribute
- For index-search-only queries (w/o retrieving the actual data records), clustered index is not needed
- Other considerations include: whether to use a hash index over a tree index (B+-tree), and whether to use dynamic hashing for the file